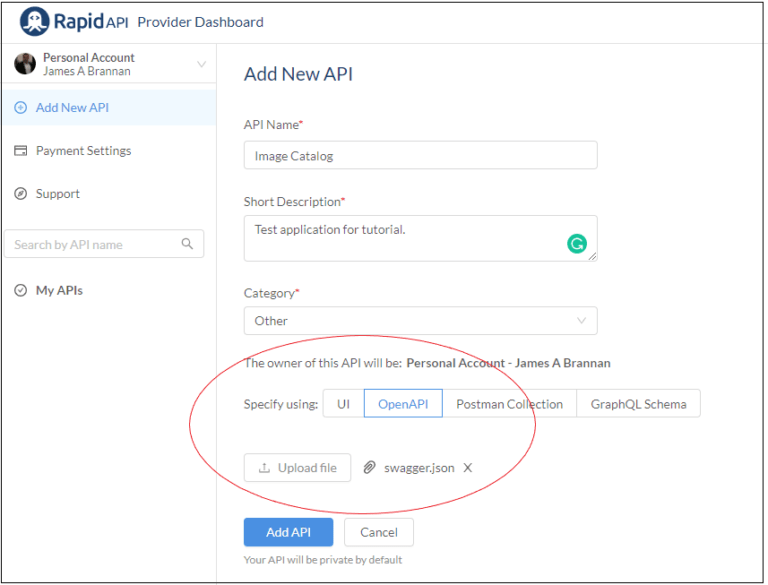
**How to Build a Java RESTful API with Spring Boot**

By [Kelly Arellano](https://rapidapi.com/blog/author/kellya/) // April 23, 2021

[**Blog**](https://rapidapi.com/blog/)**»**[**API Tutorials**](https://rapidapi.com/blog/category/tutorial/)**»**[**Java API Tutorials**](https://rapidapi.com/blog/category/tutorial/java/)**»**[**Spring Boot API Tutorials**](https://rapidapi.com/blog/category/tutorial/java/spring-boot/)**» How to Build a Java RESTful API with Spring Boot**

[**https://rapidapi.com/blog/how-to-build-an-api-with-java/**](https://rapidapi.com/blog/how-to-build-an-api-with-java/)



In this tutorial, we explore how to design and implement a RESTful API [using Java](https://rapidapi.com/blog/how-to-use-an-api-with-java/). After finishing, you should better understand the following topics:

* [Restful API Design](https://rapidapi.com/blog/20-tutorials-on-how-to-create-your-own-api-sorted-by-programming-language/),
* implementing a REST API using Java,
* documenting that REST API using Swagger,
* and publishing your API on [RapidAPI](https://rapidapi.com/).

Understanding this tutorial requires at least a cursory understanding of Java and the following Java technologies:

* JAX-RS 2.0,
* Spring Boot,
* JSON, and the
* Jackson JSON library.

However, the tutorial does not presuppose an in-depth knowledge of these technologies.

[**View the Best JavaScript APIs List**](https://rapidapi.com/collection/javascript-apis)

**Introduction**

Representational State Transfer (REST) is a web-based architectural style most developers get incorrect when implementing. Rather than creating a truly RESTful application programming interface (API), he or she implements a remote procedural call (RPC) API that proves confusing and difficult to other developers. But different pundits online have differing opinions on what is or is not truly a RESTful API. Here we will not waste time arguing what constitutes a genuinely RESTful API; instead, we design a simple RESTful API and then partially implement that API using Spring Boot and Jersey.  After implementing the API, we document the API using a tool called Swagger. Our end goal is an example of a Restful API that is easy to understand and use in client applications. To achieve this goal, we adhere to the principles of RESTful API architecture.

**REST API Design**

A RESTful architecture is an architectural style that defines how software communicates over the Internet. Client software request resources while servers respond with resources. Resources map to [universal resource identifiers (URIs)](https://rapidapi.com/blog/url-vs-url/) and actions performed on those resources map to [HTTP methods](https://rapidapi.com/blog/api-glossary/http-request-methods/) such as [POST](https://rapidapi.com/blog/api-glossary/post/), [GET](https://rapidapi.com/blog/api-glossary/get/), [PUT](https://rapidapi.com/blog/put-vs-patch/), [PATCH](https://rapidapi.com/blog/put-vs-patch/), and DELETE.

* *Google’s API Design Guide is an excellent resource for exploring REST API design (*[*API Design Guide*](https://cloud.google.com/apis/design/)*) in more depth.*

A RESTFul API provides a robust solution to the following problem.

* You have a set of resources you wish to expose to external client systems using a protocol that most modern systems understand. You want the exposure to be intuitive and inexpensive for those external systems. REST provides a means of accomplishing these business goals through resources and HTTP methods.

REST is remarkably simple at its core; however, there are many principles and best practices to follow when designing and implementing a RESTful API. But rather than discussing these principles and practices, let’s illustrate them by designing and implementing a simple hypothetical Restful API.

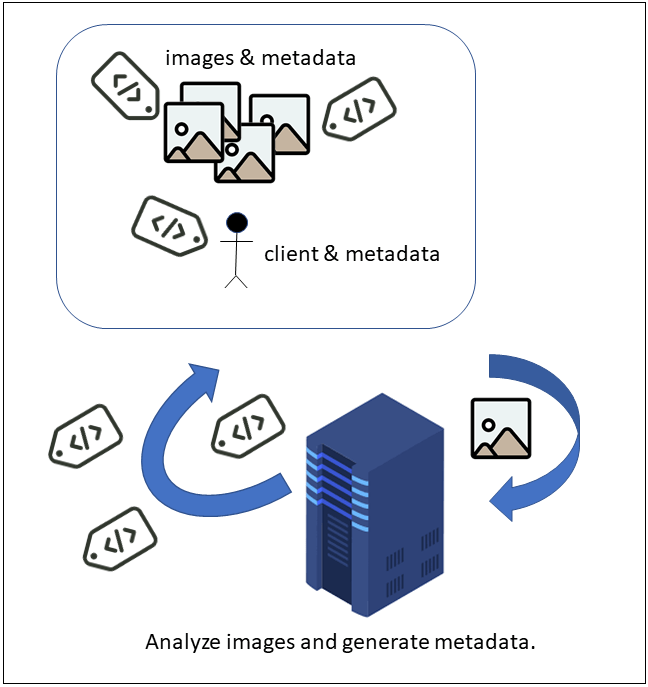
* *The Richardson Maturity Model defines what constitutes a RESTful API in more detail but not discussed here. If you would like more detail on what is or is not RESTful architecture, refer to the explanation of the Richardson Maturity Model on Martin Fowler’s website (*[*Richardson Maturity Model*](https://martinfowler.com/articles/richardsonMaturityModel.html)*).*

Design Example

Let’s create a hypothetical system to illustrate building a RESTful API.

**ClientImageCatalog**

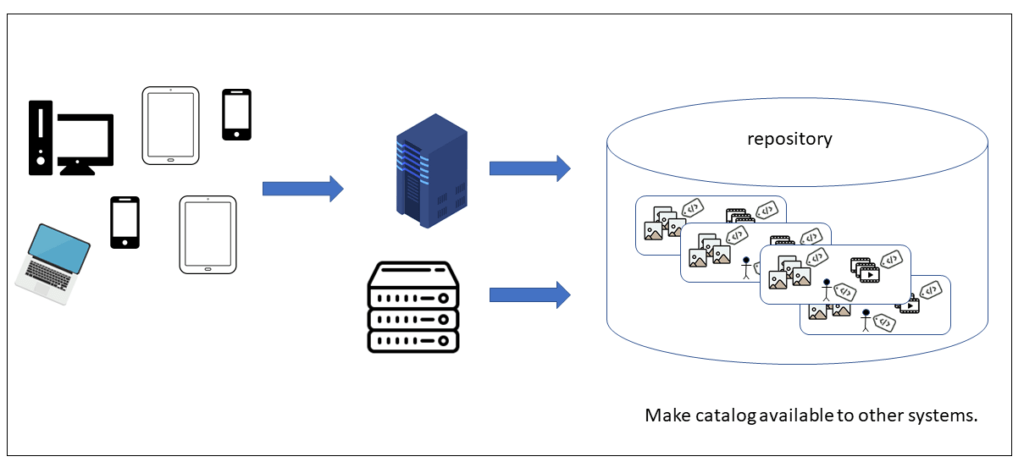
Imagine a simple system that stores images belonging to a person (termed a client) and associated generated metadata. Our system analyzes images using various tools that generate metadata describing each image. The system then stores the images and related metadata, along with information on the image’s owner (the client).



A hypothetical image analysis system.

**Public-Facing Interface**

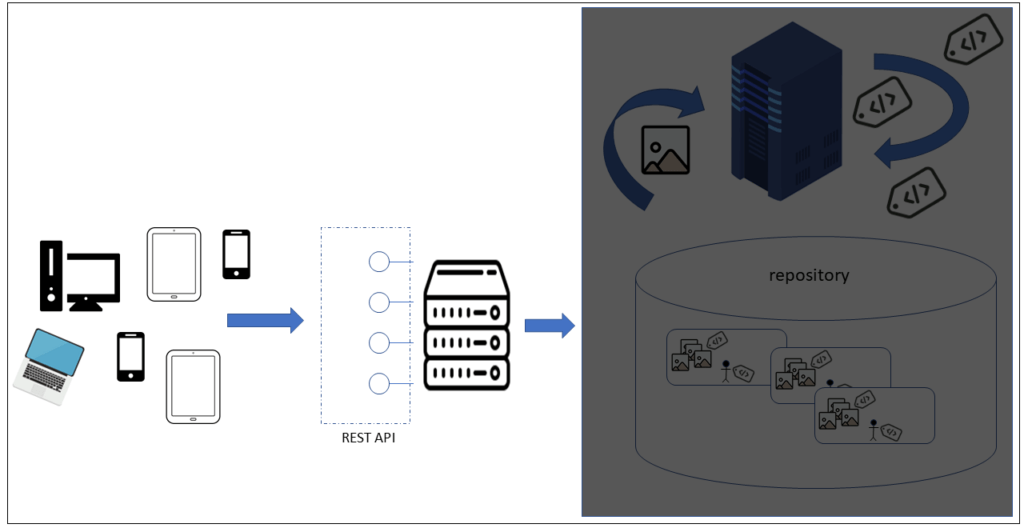
Now suppose we wish to make this system available to external systems allowing these systems to upload and manipulate images and image owners. Moreover, assume we want to make this system available to as many external systems as possible. The problem is that we have no control over these client systems, and the clients could be anything from a cellphone to another enterprise system. Attempting to code to all client systems is impossible. What we need is a simple, widely-accepted, robust API clients can use to manipulate the resources in our system. And of course, that API is a RESTful API.



Disparate clients are accessing our enterprise repository.

**Back-end System is Black-Box**

When developing software, often, a designer must abstract certain aspects of a complete system to make understanding individual portions of the system more tractable. Here we abstract our hypothetical system, as we are not concerned with the Artificial Intelligence behind analyzing images or the storage technology of storing those images. Instead, we are only concerned with how external systems will access our system’s resources. We are only concerned with creating an API client can use to access our system; we treat the more extensive system as a black-box.



The actual system is a black box.

**Generic Meta-data**

Lastly, let’s add one final bit of complexity. Assume we cannot define the metadata generated in advance. We do not know what metadata the system will generate, and we wish the system to be easily expandable to allow new metadata. We keep our API metadata agnostic to accommodate new metadata over time. Of course, there are some obvious properties of an image such as the image format, and the binary data.  But for non-obvious properties, we are restricted to a collection of key/value metadata pairs describing the image. The same restriction applies to the person (client) associated with his or her images. As an aside, by making “client” metadata agnostic, a client could be anything that has a collection of images. But for our purposes, consider a client a person.

API Design Tasks

We’ve described our hypothetical system, so let’s start designing the API for accessing it. Designing a robust REST API requires a minimum of the following activities:

* determine the resources;
* create a resource model;
* formalize the resource model as an object model;
* create JSON schemas (if using JSON) of the resources;
* write a list of actions to be performed on the resources;
* translate the object model into URLs;
* map the actions to HTTP methods and query parameters;

*\* Query parameters are parameters attached to the end of a URL that help define filter results or specify actions. For example, in the following URL, color and details are query parameters.*

**http://www.nowhere.com/foobars?color=green&details=full**

* decide how to represent the transmitted data (i.e., JSON, XML, or some other format);
* and define the schemas describing the resources.

Now, you could argue that translating an object model into URLs and writing resource schemas are development and not design tasks, and you are probably correct.  However, here we treat these tasks as design tasks. Regardless of how you classify these two tasks, though, you should perform them. Suspend disbelief and handle the tasks as design tasks.

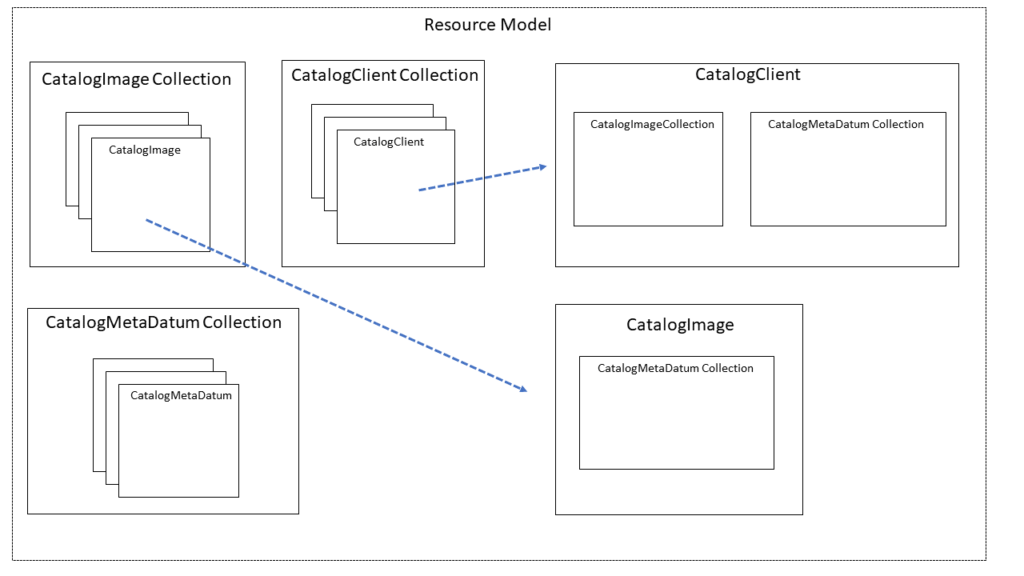
Determine Resources

Now that we know the tasks to perform, let’s begin with the first task – defining the resources. We have a client and his or her associated images.  Each client and image has associated metadata. Therefore, our resources are Client, Image, and MetaDatum. But a resource named Client, Image, or MetaDatum seems a recipe for disaster, as these are common terms and would probably confuse API consumers. Call me crazy, but calling a resource an Image appears to be asking developers using your API to confuse your image with an Image in a typical package such as java.awt.Image.  Let’s avoid any potential confusion by pre-pending “Catalog” to our resources. And so, the resources we define become CatalogClient, CatalogImage, and CatalogMetaDatum.

* CatalogClient – a person described by a collection of MetaDatum who has one or more CatalogImages.
* CatalogImage – an image described by a collection of MetaDatum that belongs to one and only one CatalogClient.
* CatalogMetaDatum – a key/value pair that expresses some aspect of a CatalogImage or CatalogClient.

Resource Model

The fundamental concept behind REST is the resource – all design centers around resources. A collection is one or more resources of the same type. A collection can have sub-collections, and a resource can have sub-resources.  These relationships between collections and resources and sub-collections and sub-resources determine your resource model.



A resource model of our system’s resources.

Our resource model is summarized as follows:

* a CatalogClient collection is one or more CatalogClient resources;
* CatalogClient resources consist of a CatalogImage collection and a CatalogMetaDatum collection;
* a CatalogImage collection is one or more CatalogImage resources and a CatalogMetaDatum collection;
* and a CatalogMetaDatum collection is one or more CatalogMetaDatum resources.

This resource model is essential in later determining the hierarchical nature of our RESTful API’s URIs for our resources. The next step is optional, we could jump to the next step, writing a list of actions, but I feel more comfortable developing an object model before defining URIs, JSON resources, and Java classes. And so, let’s translate the looseness of a resource model to a more formal object model.

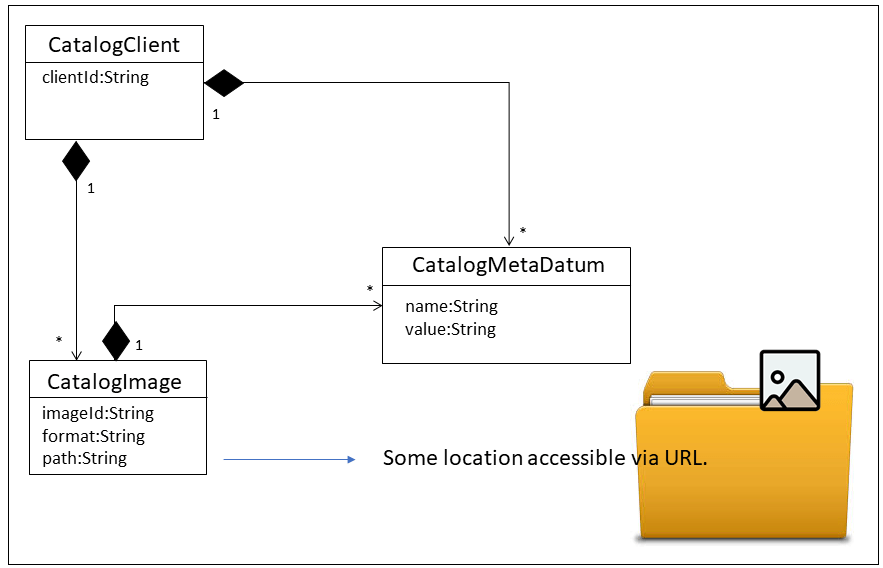
Object Model

After determining our resources and modeling them using a resource model, we model them using an object model. Here, we use the Unified Modeling Language (UML) as the notation for describing our object model. A CatalogClient has an identifier and one or more CatalogMetaDatum elements. A CatalogImage consists of an identifier, an image format, and binary.  To keep the model simple, assume a directory on a web server that stores the binary as a file. A CatalogImage also has one or more MetaDatum that describe the image. We model each resource as a Class by drawing it as a rectangle.

* *A class is synonymous with a resource.*

**Composition**

A collection of CatalogMetaDatum objects defines a CatalogClient.  These CatalogMetaDatum are dependent on the CatalogClient they express, and so their relationship is modeled using composition (solid diamond). A CatalogImage is also dependent on a CatalogClient and modeled as composition. Think of composition like this, when deleting a CatalogClient; you also remove the CatalogClient’s CatalogImage collection and CatalogMetaDatum collection. Neither object makes sense independent of its parent CatalogClient. The same is true for CatalogImage; when deleting a CatalogImage, you also delete its associated CatalogMetadatum collection.



Our system’s object model.

**Relationship Direction and Cardinality**

Note that only one side of each line modeling the composition is adorned with an arrow. What this signifies is that only a CatalogClient “knows” its CatalogImage. Moreover, only CatalogClient and CatalogImage “know” about their associated CatalogMetaDatum collection.  A number or symbol also decorates each line, indicating the relationship cardinality. A single CatalogClient contains many CatalogImages.  Both CatalogImage and CatalogClient contain many CatalogMetaDatums. These relationships become important later when we translate this object model to URIs and HTTP methods.

* CatalogClient consists of one or more CatalogImage resources and one or more CatalogMetaDatum resources. When a CatalogClient is deleted, associated CatalogImage and CatalogMetaDatum are deleted.
* CatalogImage consists of one or more CatalogMetaDatum resources. When a CatalogImage is deleted, associated CatalogMetaDatum is deleted.

Design Rules

Before continuing with our design tasks, let’s pause to consider a few best practices when developing a RESTful API. These practices will help us construct a robust yet straightforward API and help with the next design steps we undertake. The following are standard best practices espoused on many websites discussing Restful API design.

* Use nouns to represent resources, plural for collections, singular for a single resource.
* HTTP request methods define actions performed on resources.
* All resources and communication are stateless.
* Specify a version number for your API.
* Forward-slashes represent a hierarchical relationship.
* Use hyphens rather than underscores in URIs.
* Prefer lower-case letters in REST URIs.
* Never include file extensions to indicate file types in URIs.
* Use query component variables in a URI to filter collections.

**Nouns not Verbs**

Remember, REST is an architecture for requesting HTTP actions to be performed against resources. Resources are objects; nouns represent objects.  A URI is the resource’s identifier, while a URL is a resource’s identity and location. This distinction between resource and action is essential, as when developing a RESTful API, you should avoid an RPC style API.  If not sure of the difference, the following two URLs illustrate an RPC API using a verb compared to a RESTful API using a noun.

**// using a verb - RPC Style**

**http://www.nowhere.com/imageclient/getClientsById**

**// using a noun - RESTful**

**http://www.nowhere.com/imageClient/{client-id}**

This distinction between nouns and verbs will be more apparent when we determine our API’s resource URLs.

* *Note that we use the terms URL and URI interchangeably. Although technically different, as a URI, is only an identifier and not a location while a URL is both (hence URI and URL), here we treat both terms as synonymous.*

**HTTP Request Methods Define Actions**

Use HTTP request methods to manipulate resources. Do not define your own. The following table lists the most commonly used HTTP request methods.

|  |  |
| --- | --- |
| HTTP Request Methods | |
| GET | Retrieve a resource representation. |
| POST | Create a new resource. |
| PUT | Fully update an existing resource. |
| PATCH | Modify an existing resource. |
| DELETE | Delete an existing resource. |
| HEAD | Receive resource metadata. |
| OPTIONS | Receive supported HTTP request methods. |

What this means is that the same URL that identifies a resource can perform different actions on that resource. Consider the difference the HTTP request method makes to the same URL.

|  |  |
| --- | --- |
| HTTP Request Methods defining action | |
| GET   /images/<image-id> | Get the image with a particular identifier. |
| PATCH   /images/<image-id> | Update the image with the particular identifier. |
| PUT   /images/<image-id> | Update by overwriting the image with the particular identifier. |
| DELETE    /images/<image-id> | Delete the image with a particular identifier. |

If you are accustomed to developing RPC style APIs, then using an HTTP request method to distinguish between the action to take on a resource might seem confusing. However, it will become more intuitive as you become comfortable with RESTful API design.

**Corollary Antipattern**

Do not do the following, as this is a common antipattern when developing a RESTful API. It might seem easy to succumb to the seductive convenience of turning a RESTful API into an RPC API, but I promise, it only leads to confusion later. An RPC-style “RESTful API” such as the following, is harder for developers of external systems to understand and makes your API is harder to maintain. The creators of the HTTP methods are most likely smarter than you or I, trust their judgment and stick to the HTTP methods.

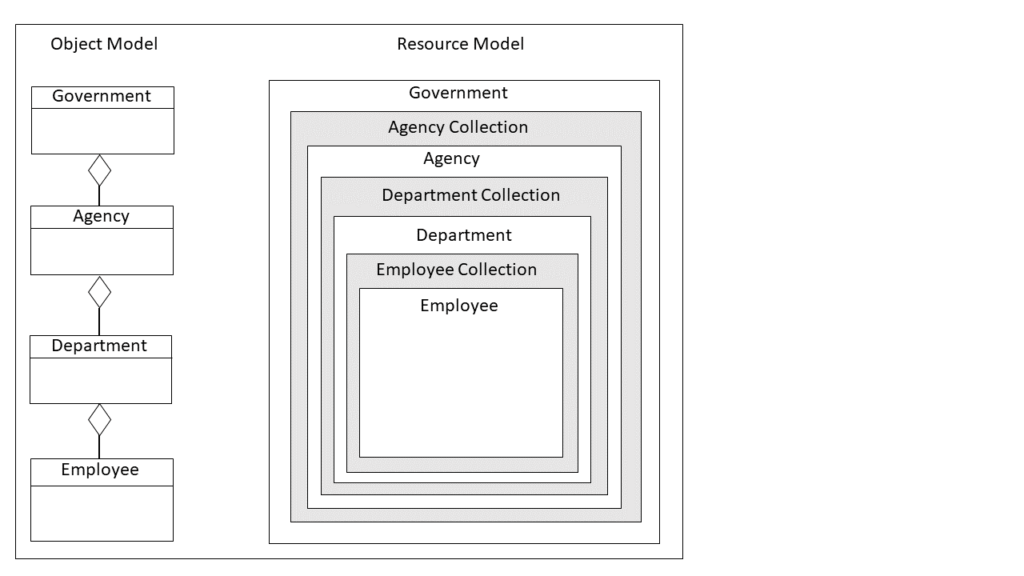
|  |
| --- |
| REST as RPC (*Do not do this – it is an antipattern*) |
| GET  http://www.nowhere.com/images/createImage?imageId=123 |
| GET http://www.nowhere.com/images/getImage?imageId=123 |
| GET http://www.nowhere.com/images/updateImage?imageId=123 |
| GET http://www.nowhere.com/images/replaceImage?imageId=123 |
| GET http://www.nowhere.com/images/deleteImage?imageId=123 |

**Resources Must be Stateless**

Resources must be stateless. If familiar with more traditional web development, then this might seem a problematic edict. Yes, it requires not using cookies, URL parameters, and session variables to maintain state. But consider this, these constructs are all schemes to try avoiding the harsh reality of the web, namely, that the Internet is stateless. But REST embraces the statelessness of the Internet and admits that these schemes all try to circumvent this reality of statelessness. So if you are designing a RESTful API, design it as a stateless system. Of course, a client sometimes requires that it maintain state, but if it does, then the client and not the server should maintain state. A server must know nothing of a client’s statefulness. A RESTful API is stateless.

**Represent Resources as Hierarchical Relationships**

Resources are hierarchical.  REST interfaces should reflect this hierarchy. A well-designed hierarchical path makes your API easy to understand.  The following object model and its translation to URLs illustrates a RESTful hierarchy.



A hierarchical object relationship between Agency, Department, and employee.

We can translate these resources, and their relationships, into a hierarchy,

* a government is a collection of agency resources,
* an agency is a collection of department resources,
* and a department is a collection of employee resources.

We can translate the object model just created into the hierarchical URIs in the following table.

|  |
| --- |
| REST Hierarchical URIs From Object Model |
| /government |
| /government/agencies |
| /government/agencies/<agency-id> |
| /government/agencies/<agency-id>/departments |
| /government/agencies/<agency-id>/departments/<dept-id> |
| /government/agencies/<agency-id>/departments/<dept-id>/employees |
| /government/agencies/<agency-id>/departments/<dept-id>/employees/<emp-id> |

**Specify a Version**

Well designed REST APIs should include a version. Imagine the horror clients of your API will have if you were to suddenly change your REST API; all systems consuming your REST API endpoints would break. To avoid this problem, version your application so your APIs can continue to work with your old API and then orderly transition to your new API. The four common methods to version your API are to

* add the version to the URI,
* create a custom request header,
* append a query parameter,
* and change the accept header.

**Versioning in URI**

One technique for versioning is to add it directly to your API’s URL.  For example, the following illustrates using versioning in an API’s URL.

|  |
| --- |
| REST Versioning in URL |
| /v1/government |
| /v1/government/agencies |
| /v1/government/agencies/<agency-id> |
| /v2/government |
| /v2/governement/agencies |
| /v2/governement/agencies/<agency-id> |

When you use the version number in the URL, the client is requesting a different resource. Some claim using a version in the URL is technically violating RESTful architecture, as a URI is supposed to be a resource identifier, and adding a version breaks that contract. However, using a URI with a version is a common and pragmatic means of versioning a RESTful API.

**Versioning in Header**

Another technique for versioning is to add the version to the Http header of your HTTP request as a custom header, or using an accept header. These are also common techniques; for example, you might add a header that has the following value.

**Accept-API-Version: resource=2.0**

Versioning can be a contentious topic when discussing RESTful APIs. But the answer is to pragmatically research and choose the technique most useful to your particular circumstance.

* Versioning using custom Accept header ([Versioning REST APIs with a custom Accept header in Jersey](https://codebias.blogspot.com/2014/03/versioning-rest-apis-with-custom-accept.html)).
* Versioning using version in URI ([Versioning web-services with Java EE](https://ecolabardini.github.io/2015/11/23/versioning-web-services-with-java-ee/)).
* List of versioning techniques used by major websites ([How are REST APIs versioned?](http://www.lexicalscope.com/blog/2012/03/12/how-are-rest-apis-versioned/)).

For most of this tutorial, we do not version our API; however, we will cover an easy technique you can use to version your API if your API doesn’t change often.

JSON Data Representation

Enough discussion of best practices, let’s return to design tasks by defining our resources using JSON schemas. As the object-model above illustrates, we need to create a CatalogClient, CatalogMetaDatum, and CatalogImage resource. Moreover, we must make these resources available to and understandable by external systems. To facilitate this communication, we will transmit our resources as JSON objects. But how does an external system know the format of these resources? Simple, a JSON schema. Let’s define the JSON schema representation of our three resources.

**CatalogImage and CatalogMetaDatum Schema**

Admittedly, there is no standard admonishing developers to develop a JSON schema before using JSON objects. But in my opinion, one valuable concept JSON adoption lost when it replaced XML in popularity was explicitness. An XML schema is a powerful concept, and it fully defines a system’s object model in an easy to read and formal, albeit rather wordy, format.  The loose nature of JSON adoption by most developers often loses this explicitness. Developers other than the developer defining the JSON resource are often left to “figure out” the resource’s structure through trial and error. But a schema helps avoid this trial and error.  And so, although not required, creating JSON schemas provide a human and machine-readable description of resources that makes your REST API more robust. And, as you see in a few sections, machine-readable descriptions allow for machine-generated code.

**JSON Schema**

JSON schema allows JSON document validation. It provides an explicit description of your API’s JSON data. JSON Schema has never been formally accepted as a standard as XML Schema was ([W3C XML Schema](https://www.w3.org/standards/xml/schema)). Instead, it remains as a draft ([JSON Schema](https://json-schema.org/specification.html)) and not an agreed specification. But enough developers are using the JSON Schema specification to make it useful.

* *For more information on JSON schemas, refer to the W3C site, here’s an excellent tutorial (*[*JSON Schemas*](https://www.w3schools.in/json/json-schemas/)*).*

**CatalogImage JSON Schema**

* Create a new file named CatalogImage.schema.json and copy the following to the file.

**{**

**"$schema": "http://json-schema.org/draft-04/schema#",**

**"title": "CatalogImage",**

**"description": "A image from Image catalog",**

**"type": "object",**

**"properties": {**

**"image-id": {**

**"description": "The unique identifier for an image.",**

**"type": "string"**

**},**

**"image-format": {**

**"description": "Format of image (gif, png, jpeg, etc.).",**

**"type": "string"**

**},**

**"image-path": {**

**"description": "Path/URL to the image data.",**

**"type": "string"**

**},**

**"meta-data": {**

**"description": "Metadata item describing resource.",**

**"type": "array",**

**"items": {**

**"type": "object",**

**"title": "CatalogMetaDatum",**

**"description": "The meta data object comprising the array.",**

**"properties": {**

**"name": {**

**"type": "string",**

**"description": "Meta data element property name."**

**},**

**"value": {**

**"type": "string",**

**"description": "Meta data element property value."**

**}**

**}**

**}**

**}**

**}**

**}**

* Save the file somewhere accessible so you can move it to your project later.

The schema’s top-level element is CatalogImage of type object.  That object has an image-id, image-format, image-path, and meta-data property. The meta-data property is an array of objects of type CatalogMetaDatum. The CatalogMetaDatum consists of a name and a value.

The schema defines each element’s type using the type attribute. In our schema, we have the types: string and object; however, there are other types you can use. The basic schema types are:

* string,
* number,
* integer,
* boolean,
* and null.

For more information on types, refer to the JSON Schema website ([Basic Types](https://cswr.github.io/JsonSchema/spec/basic_types/)).

Other schema elements included in our schema are a title (the user-friendly element name) and description, which allows adding human-readable documentation. Again, you should refer to the JSON Schema website for a more complete description of JSON schemas ([JSON Schema](https://cswr.github.io/JsonSchema/)).

**CatalogClient JSON Schema**

In this tutorial, we treat CatalogClient as an exercise, and do not define a CatalogClient structure. However, we do require a bare-bones definition, so our code compiles in the following steps.

* Save the following simplistic schema to the same folder as CatalogClient.schema.json.

**{**

**"$schema": "http://json-schema.org/draft-04/schema#",**

**"title": "CatalogClient",**

**"description": "A Client from Image catalog",**

**"type": "object"**

**}**

Modeling The Resource URIs

Now that we have defined the JSON schemas for the resources (CatalogClient, CatalogImage, and CatalogMetaDatum), let’s define the REST API by first describing in complete English sentences (or whatever your native tongue happens to be) the actions external clients will perform with our system’s provided resources. Note that we switch to referring to the resources as clients, images, and metadatum rather than CatalogClient, CatalogImage, or CatalogMetaDatum for simplicity.

**Write Interactions Using Sentences**

Writing the interactions to be performed on our resources are essential to understand before we start coding our API. Usually, create, replace, update, and delete (CRUD) is a good starting point when considering the actions an API must support.

* Write each interaction that an external client will perform with our API. Number them so you can refer to them by their number in later design steps.

1. Return a client with the specified client-id.
2. Return an image with the specified image-id.
3. Respond with all clients who match the supplied metadata.
4. Get all images that match the metadata provided.
5. Return images that match the provided image metadata for the client with the specified client-id.
6. Create a new client.
7. Create a new image for the client with the specified client-id.
8. Fully update (overwrite) the client with the specified client-id.
9. Fully update (overwrite) the image with the specified image-id.
10. Update a client with the specified client-id.
11. Update an image with the specified image-id.
12. Delete a client with the specified client-id.
13. Delete an image with the specified image-id.

**Transform Sentences into URLs**

After understanding the resources, the resources relationships, and how those resources will be manipulated, we translate our analysis into a list of REST endpoints (URLs) and corresponding HTTP commands.

* Refer to the table of HTTP Commands and the resource modeling to determine each URL.
* Create five GET requests to retrieve the resources.
* Write two POST requests to create the resources.
* Add two PUT requests to replace the resources.
* Create two PATCH requests to update the resources.
* Write two DELETE requests to delete the resources.

**URI Templates**

|  |  |  |
| --- | --- | --- |
| Resource URI Templates | | |
| 1. | GET | /clients/{client-id} |
| 2. | GET | /images/{image-id} |
| 3. | GET | /clients?(meta-data=<name>=<value>&)\* |
| 4. | GET | /images?(meta-data=<name>=<value>&)\* |
| 5. | GET | /clients/{client-id}/images?(meta-data<name>=<value>&)\* |
| 6. | POST | /clients      *\*body includes <client-json-data)>* |
| 7. | POST | /clients/{client-id}/images      *\*body includes  <image-json-data> && <binary-data>* |
| 8. | PUT | /clients/{client-id}     *\*body includes <client-json-data>* |
| 9. | PUT | /images/{image-id}      *\*body includes <image-json-data> && <binary-data>* |
| 10. | PATCH | /clients/{client-id}      *\*body includes <client-json-data>* |
| 11. | PATCH | /images/{image-id}     *\*body includes <image-json-data>* |
| 12. | DELETE | /clients/{client-id} |
| 13. | DELETE | /images/{image-id} |
| * <x> – notation, signifies a literal value * ()\* – repeat statements in parenthesis n times | | |
| * *A URI template is a sequence of characters for describing a URI through variable expansion (*[*URI Template definition*](https://tools.ietf.org/html/rfc6570)*). For example {client-id}, through variable expansion becomes “123”, or some other particular value.* | | |

The numbered URI templates correspond to the numbered statements we wrote earlier in the analysis. We use these numbers again when implementing our API in Java. Here too, we refer to our resources using the original terms, there is little danger in misinterpreting the client, image, or metadatum in a URL, and so we use the more compact name to keep the URI templates shorter. This decision is arbitrary, if you disagree with the decision, suspend disbelief for now and continue with the tutorial.

**Hierarchy**

The hierarchy is intuitive; however, we do pragmatically violate the hierarchy in a few instances.  First, we allow obtaining images independently of their parent clients. Second, we do not require knowing a client-id and an image-id to obtain a particular image for a specific client. The image-id alone is sufficient. However, the hierarchical nature of clients and images is apparent when you consider the following URL template, as images belong to clients in this statement.

* /clients/{client-id}/images?(meta-data<name>=<value>&)\*

Obtaining a client or an image is a GET operation while obtaining a specific client or specific image requires adding its identifier to the URI as a URI template parameter. Obtaining a client or image collection requires filtering the resource collection using one or more metadatum key/value pairs. Note that we repeat the query parameter’s name for each key/value pair for the parameter when passing a list as a query parameter’s value. For example, obtaining all portraits of females between the ages of 30 and 50 might appear similar to the following URL.

**http://nowhere.com/images?meta-data=face-identified=true&meta-data=setting=portrait&meta-data=gender=female&meta-data=age=30-50**

The meta-data keys are face-identified, setting, gender, and age, while the values are true, portrait, female, and 30-50. To pass them as a list of values of the meta-data query parameter we pass the key/value as a single statement (setting=portrait)

**Coding The API**

Now that we have designed our RESTful API, we can start implementing it. In the real world, deciding what to use to implement your API might prove a lengthy and possibly political process. But here, let’s use Spring Boot with Jersey (JAX-RS 2.0) to implement our API. Note that we only partially implement the API, just enough to illustrate how to translate our design into code.

Although you can use many different programming languages and [frameworks](https://rapidapi.com/blog/api-glossary/api-framework/) to program a RESTful API; here, we use Java with Spring Boot and the Jersey framework. We use Eclipse Jersey because it implements the JAX-RS API rather than Spring’s non-standard MVC implementation of REST. Jersey is an opensource framework for developing RESTful Webservices and is the reference implementation for the Java API for RESTful Web Services (JAX-RS) specification. The JAX-RS specification is the accepted industry standard for Java REST API development. For a more in-depth discussion of JAX-RS and the importance of standards, refer to the article [Top Java Rest Frameworks](https://rapidapi.com/blog/top-java-rest-frameworks/).

* The Eclipse Jersey website contains more information on Jersey ([Eclipse Jersey](https://eclipse-ee4j.github.io/jersey/)).

JAX-RS 2.0

JAX-RS is a specification consisting of interfaces and annotations. It is part of the JEE specification in the package javax.ws.rs, which contains the JAX-RS annotations and interfaces ([package javax.ws.rs](https://docs.oracle.com/javaee/7/api/javax/ws/rs/package-summary.html)). Vendors implement the specification by providing implementations of the provided interfaces and annotations. It is particularly crucial that you understand the JAX-RS annotations, as this is how we map the URIs we identified to Java methods.

**Annotations**

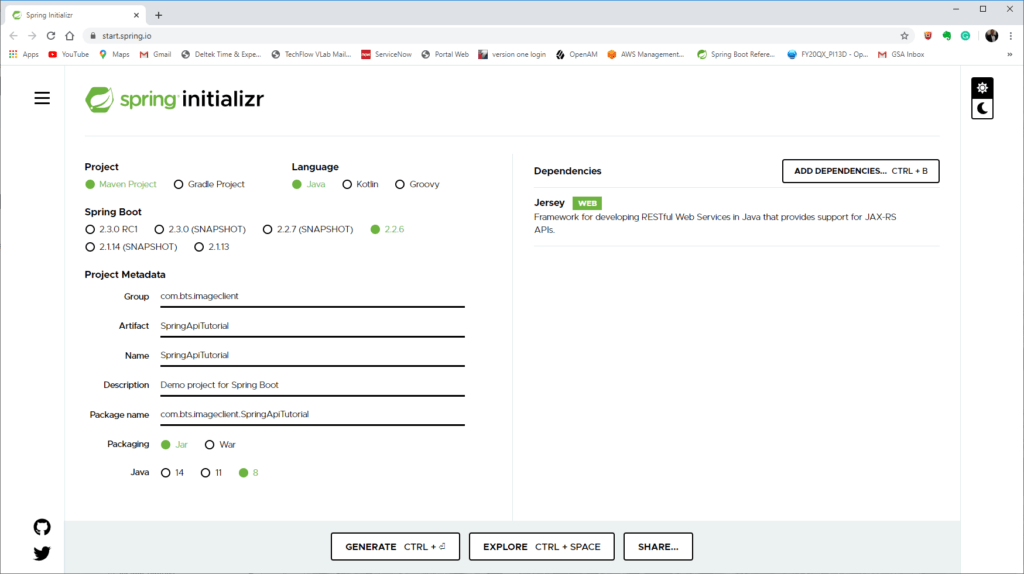
You use JAX-RS in your application by annotating classes with JAX-RS annotations. When compiling your project, these annotations are interpreted by the JAX-RS implementation (in this tutorial Jersey) and translated into the appropriate Java code. Refer to the specification for a full listing and description of the available annotations ([JAX-RS: Java API for RESTful Web Services](http://jcp.org/en/jsr/detail?id=370)); here we use the @Path, @POST, @GET, @PUT, @PATCH, @DELETE, @PathParam, @QueryParam, and @Produces annotations. The following illustrates the annotation followed by its JavaDoc link and a usage example.

|  |  |
| --- | --- |
| Annotations | |
| @Path(“<resource path>”) | Specifies the URI path relative to a resource. |
| [javax.ws.rs.PATH](https://docs.oracle.com/javaee/7/api/javax/ws/rs/Path.html) | |
| **import *javax.ws.rs.Path*;**  **...**  **@Path("/clients")**  **public class ClientController** | |
| @POST | Specifies that a method handles HTTP POST requests. |
| [javax.ws.rs.POST](https://docs.oracle.com/javaee/7/api/javax/ws/rs/POST.html) | |
| **import *javax.ws.rs.POST*;**  **...**  **@POST**  **public Response createClient(Client client)** | |
| @PUT | Specifies that a method handles HTTP PUT requests. |
| [javax.ws.rs.PUT](https://docs.oracle.com/javaee/7/api/javax/ws/rs/PUT.html) | |
| **import *javax.ws.rs.PUT*;**  **...**  **@PUT**  **@Path("{client-id}")**  **public Response updateClient(@PathParam("client-id") Integer clientId, Client client)** | |
| @GET | Specifies that a method handles HTTP GET requests. |
| [javax.ws.rs.GET](https://docs.oracle.com/javaee/7/api/javax/ws/rs/GET.html) | |
| **import *javax.ws.rs.GET*;**  **...**  **@GET**  **@Path("/{client-id}")**  **public Response getClient(@PathParam("client-id") Integer clientId)** | |
| @DELETE | Specifies a method that handles HTTP DELETE requests. |
| [javax.ws.rs.DELETE](https://docs.oracle.com/javaee/7/api/javax/ws/rs/DELETE.html) | |
| **import *javax.ws.rs.DELETE*;**  **...**  **@DELETE**  **@Path("/{client-id}")**  **public Response deleteClient(@PathParam("client-id") Integer clientId)** | |
| @PathParam(“<parameter name>”) | Inject resource identifier from the URL as a method parameter. |
| [javax.ws.rs.PathParam](https://docs.oracle.com/javaee/7/api/javax/ws/rs/PathParam.html) | |
| **import *javax.ws.rs.PathParam*;**  **...**  **@GET**  **@Path("/{image-id}")**  **public Response getImageById(@PathParam("image-id") Integer imageId** | |
| @Produces | Define the MIME type returned from a resource method. |
| [javax.ws.rs.Produces](https://docs.oracle.com/javaee/7/api/javax/ws/rs/Produces.html) | |
| **import *javax.ws.rs.Produces*;**  **...**  **@Path("/clients")**  **@Produces("application/json")**  **public class ClientController** | |
| @Consumes | Define the MIME type consumed by a resource method. |
| [javax.ws.rs.Consumes](https://docs.oracle.com/javaee/7/api/javax/ws/rs/Consumes.html) | |
| **import *javax.ws.rs.Consumes*;**  **...**  **@POST**  **@Consumes("application/json")**  **public createClient(Client client)** | |
| @QueryParam | Bind query parameters to method parameters |
| [javax.ws.rs.QueryParam](https://docs.oracle.com/javaee/7/api/javax/ws/rs/QueryParam.html) | |
| **import *javax.ws.rs.QueryParam*;**  **...**  **@GET**  **public Response getClients(@QueryParam("first-name") String firstName, @QueryParam("last-name") String lastName)** | |

Spring Boot Project Setup

If new to Spring Boot, there are numerous online resources for learning the framework ([Building an Application with Spring Boot](https://spring.io/guides/gs/spring-boot/)). Here we spend minimal time discussing the framework, instead only using it to illustrate implementing our REST API.

* Navigate to the [start.spring.io](https://start.spring.io/) website.
* Add the Jersey framework to the project’s dependencies.
* Add the proper settings and generate the project as an executable jar file.

Project setup in Spring Initializr.

* Download the generated project and unzip it to the desired location.

The generated POM file should appear as the POM listed below. Notice there are no apparent dependencies on Jersey nor Jackson; the Jersey dependencies, including the JSON library Jackson, are aggregated into the larger spring-boot-starter-jersey jar.

**<?xml version="1.0" encoding="UTF-8"?>**

**<project xmlns="http://maven.apache.org/POM/4.0.0"**

**xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">**

**<modelVersion>4.0.0</modelVersion>**

**<parent>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-parent</artifactId>**

**<version>2.2.6.RELEASE</version>**

**<relativePath /> <!-- lookup parent from repository -->**

**</parent>**

**<groupId> com.bts.imageclient</groupId>**

**<artifactId>SpringApiTutorial</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**<name>SpringApiTutorial</name>**

**<description>Creating a Rest API Tutorial Project</description>**

**<properties>**

**<java.version>1.8</java.version>**

**</properties>**

**<dependencies>**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-jersey</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-test</artifactId>**

**<scope>test</scope>**

**<exclusions>**

**<exclusion>**

**<groupId>org.junit.vintage</groupId>**

**<artifactId>junit-vintage-engine</artifactId>**

**</exclusion>**

**</exclusions>**

**</dependency>**

**</dependencies>**

**<build>**

**<plugins>**

**<plugin>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-maven-plugin</artifactId>**

**</plugin>**

**</plugins>**

**</build>**

**</project>**

The SpringApiTutorialApplication.java file generated should appear as follows.

**package *com.bts.imageclient*;**

**import *org.springframework.boot.SpringApplication*;**

**import *org.springframework.boot.autoconfigure.SpringBootApplication*;**

**@SpringBootApplication**

**public class SpringApiTutorialApplication {**

**public static void main(String[] args) {**

**SpringApplication.run(SpringApiTutorialApplication.class, args);**

**}**

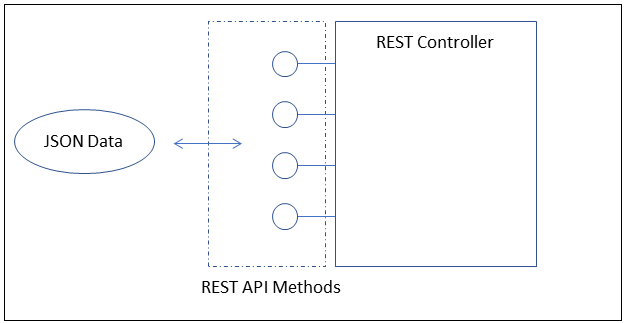
**}**

* Compile the project using Maven just to ensure everything is correct.

After the project compiles successfully, we can start coding our REST API.

Coding the Controllers

You can call the REST API classes controllers, services, or whatever you wish to name them; however, here we call them controllers. As an aside, my preference for the term controller probably stems from Spring MVC, which refers to these classes as controllers by using the @Controller Spring annotation. The critical point, though, is that these classes are how client applications access our REST API. The controllers will contain the logic needed to make our resources available via HTTP.

REST API controller.

The controller houses the methods (represented by lollipops) that clients use to manipulate the resources our REST API provides. A client can communicate with those endpoints by using a variety of data types, although XML or JSON is the data types most commonly used. Our API will use JSON exclusively.

* We do not implement the endpoints that send and receive the binary image data, see the tutorial [How to Use an API using a REST Template](https://rapidapi.com/blog/how-to-use-an-api-with-spring-resttemplate/) for examples of uploading and downloading binary data.

The REST API endpoints in our API will take JSON resources as input and return JSON resources as output; therefore, before coding our controllers, we should implement our resources as Plain Old Java Classes (POJOs). From the object model, we already know our REST payload; it consists of CatalogImage, CatalogClient, and CatalogMetadata objects.  Let’s define these resources.

**JSON Data Objects**

JSON is a data-interchange format that uses JavaScript syntax to describe data objects that consist of key/value pairs. Understanding JSON is a prerequisite for understanding this tutorial.

* A good starting point, if unfamiliar with JSON, is the Wikipedia page ([JSON](https://en.wikipedia.org/wiki/JSON)).

In this tutorial, we assume JSON as the data format used for requests and responses.

**Jackson JSON Library**

Mapping JSON to Java objects requires a library to perform the serialization/deserialization of resources. Jersey supports Jackson by default, and the spring-boot-starter-jersey package includes the needed libraries, so using Jackson in a Spring Boot Jersey application requires no configuration. Understanding Jackson, other than it is used to serialize/deserialize between JSON and POJOs automagically, is not necessary to continue with the tutorial.

* If you wish to explore Jackson further, refer to the following tutorial ([Jackson JSON Java Parser API Example Tutorial](https://www.journaldev.com/2324/jackson-json-java-parser-api-example-tutorial)).

One thing a library such as Jackson makes possible is automatic code generation. For example, if we wished to create our POJOs from a JSON schema, then a tool could automate this for us using the Jackson library. One tool freely available online is the jsonschema2pojo tool.

**The jsonschema2pojo Maven Plugin**

Let’s return to the two JSON schemas created earlier and generate our application’s POJOs using the jsonschema2pojo online tool ([jsonschema2pojo.org](http://www.jsonschema2pojo.org/)). If interested in using the tool interactively, then refer to the [How To Use an API with Spring RestTemplate](https://rapidapi.com/blog/how-to-use-an-api-with-spring-resttemplate/) tutorial; here, we use the tool’s Maven plugin. The instructions for using this plugin are available on the project’s website ( [jsonschema2pojo Maven plugin](https://github.com/joelittlejohn/jsonschema2pojo/wiki/Getting-Started#the-maven-plugin)).

* Add the following plugin to the plugins section of the project’s pom.xml file.

**<plugin>**

**<groupId>org.jsonschema2pojo</groupId>**

**<artifactId>jsonschema2pojo-maven-plugin</artifactId>**

**<version>1.0.2</version>**

**<configuration>**

**<sourceDirectory>${basedir}/src/main/resources/schema</sourceDirectory>**

**<targetPackage>com.bts.imageclient.rest.api.types</targetPackage>**

**<useTitleAsClassname>true</useTitleAsClassname>**

**<includeAdditionalProperties>false</includeAdditionalProperties>**

**</configuration>**

**<executions>**

**<execution>**

**<goals>**

**<goal>generate</goal>**

**</goals>**

**</execution>**

**</executions>**

**</plugin>**

The plugin’s configuration settings specify the source directory, that the schema’s title generates the class names, and that the generated class should not generate an additional properties property and associated getter and setter.

* Create a schema folder under the project’s resources folder.
* Add the two schemas to this folder.

**--SpringAPITutorial**

**|**

**|-- src**

**|**

**|-- resources**

**|**

**|-- schema**

**|**

**|-- CatalogClient.schema.json**

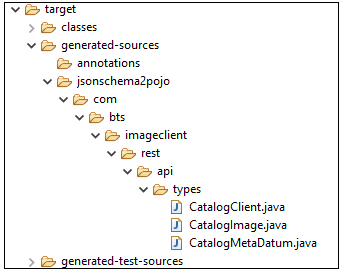
**|**

**|-- CatalogImage.schema.json**

* Execute Maven and specify the generate-sources life-cycle phase task.

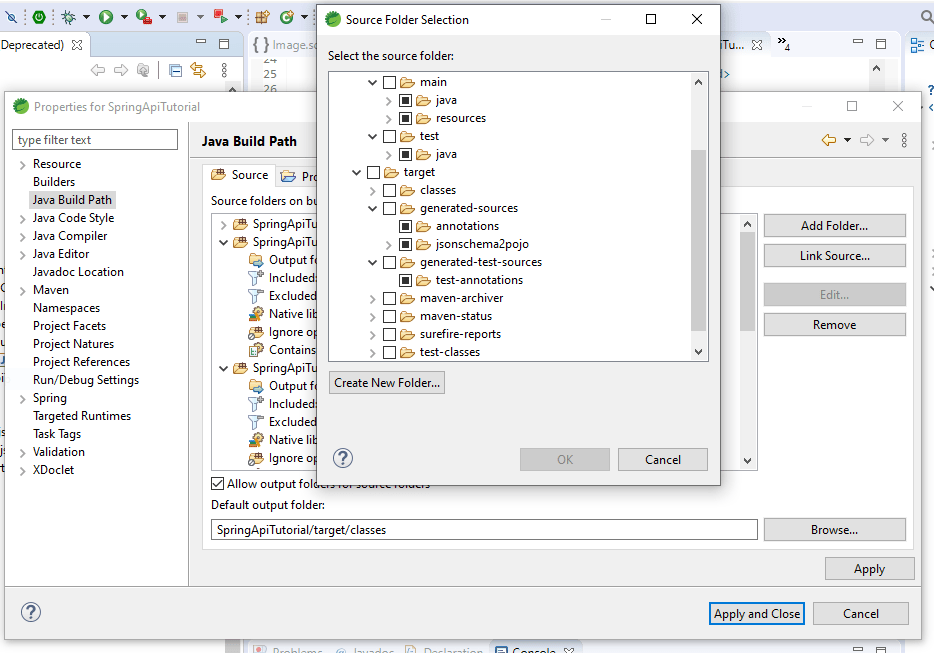
**> mvn generate-sources**

* Notice that Maven creates the CatalogClient, CatalogImage, and CatalogMetaDatum classes in the generated-sources sub-folder in target.

Generated POJOs from JSON schema.

* Either add this newly created path to your project’s classpath, or move the files manually to the correct location in your project’s source folder.

I use the Spring Tool Suite 4 (Eclipse), and so I modified my project’s classpath to include the generated types folder. Your IDE will doubtless have a way to add this folder to your project’s classpath.

Adding a folder to the classpath in Spring Tool Suite 4.

**Never modify the generated classes directly, change the classes by modifying the classes’ JSON Schema, and then executing the Maven generate task.**

Let’s examine the three generated classes, starting with the CatalogMetaDatum class.

**CatalogMetaDatum POJO**

The CatalogMetaDatum class contains the two properties defined in the CatalogImage’s JSON schema and creates getters and setters for the properties.

**package *com.bts.imageclient.rest.api.types*;**

**import *com.fasterxml.jackson.annotation.JsonInclude*;**

**import *com.fasterxml.jackson.annotation.JsonProperty*;**

**import *com.fasterxml.jackson.annotation.JsonPropertyDescription*;**

**import *com.fasterxml.jackson.annotation.JsonPropertyOrder*;**

**/\*\***

**\* CatalogMetaDatum**

**\* <p>**

**\* The meta data object comprising the array.**

**\***

**\*/**

**@JsonInclude(JsonInclude.Include.NON\_NULL)**

**@JsonPropertyOrder({**

**"name",**

**"value"**

**})**

**public class CatalogMetaDatum {**

**/\*\***

**\* Meta data element property name.**

**\***

**\*/**

**@JsonProperty("name")**

**@JsonPropertyDescription("Meta data element property name.")**

**private String name;**

**/\*\***

**\* Meta data element property value.**

**\***

**\*/**

**@JsonProperty("value")**

**@JsonPropertyDescription("Meta data element property value.")**

**private String value;**

**/\*\***

**\* Meta data element property name.**

**\***

**\*/**

**@JsonProperty("name")**

**public String getName() {**

**return name;**

**}**

**/\*\***

**\* Meta data element property name.**

**\***

**\*/**

**@JsonProperty("name")**

**public void setName(String name) {**

**this.name = name;**

**}**

**/\*\***

**\* Meta data element property value.**

**\***

**\*/**

**@JsonProperty("value")**

**public String getValue() {**

**return value;**

**}**

**/\*\***

**\* Meta data element property value.**

**\***

**\*/**

**@JsonProperty("value")**

**public void setValue(String value) {**

**this.value = value;**

**}**

**@Override**

**public String toString() {**

**StringBuilder sb = new StringBuilder();**

**sb.append(CatalogMetaDatum.class.getName()).append('@').append(Integer.toHexString(System.identityHashCode(this))).append('[');**

**sb.append("name");**

**sb.append('=');**

**sb.append(((this.name == null)?"<null>":this.name));**

**sb.append(',');**

**sb.append("value");**

**sb.append('=');**

**sb.append(((this.value == null)?"<null>":this.value));**

**sb.append(',');**

**if (sb.charAt((sb.length()- 1)) == ',') {**

**sb.setCharAt((sb.length()- 1), ']');**

**} else {**

**sb.append(']');**

**}**

**return sb.toString();**

**}**

**@Override**

**public int hashCode() {**

**int result = 1;**

**result = ((result\* 31)+((this.name == null)? 0 :this.name.hashCode()));**

**result = ((result\* 31)+((this.value == null)? 0 :this.value.hashCode()));**

**return result;**

**}**

**@Override**

**public boolean equals(Object other) {**

**if (other == this) {**

**return true;**

**}**

**if ((other instanceof CatalogMetaDatum) == false) {**

**return false;**

**}**

**CatalogMetaDatum rhs = ((CatalogMetaDatum) other);**

**return (((this.name == rhs.name)||((this.name!= null)&&this.name.equals(rhs.name)))&&((this.value == rhs.value)||((this.value!= null)&&this.value.equals(rhs.value))));**

**}**

**}**

Notice that the jsonschema2pojo tool added Jackson Annotations automatically despite not needing the annotations. For example, you typically use the @JsonProperty to mark non-standard getter/setter methods for a JSON property. If a property has a getter and setter of the same name with the prefixes get and set attached to the property name, then the @JsonProperty annotation is not needed. But if a POJO does not follow this pattern, then the @JsonProperty is used to bypass the property naming limitation. The following code snippet illustrates the difference.

**// ================= standard property =====================**

**String myStandardProperty;**

**public void setMyStandardProperty(String myStandardProperty) {**

**this.myStandardProperty = myStandardProperty;**

**}**

**public String getMyStandardProperty(){**

**return myStandardProperty;**

**}**

**// ================== non-standard property ==================**

**@JsonProperty("myProperty")**

**String myProperty;**

**@JsonProperty("myProperty")**

**public void setTheProperty(String theProperty) {**

**myProperty = theProperty;**

**}**

**@JsonProperty("myProperty")**

**public String getTheProperty(){**

**return myProperty;**

**}**

The property myStandardProperty, because it follows the naming convention of capitalized property name prefixed by get and set, Jackson automatically recognizes that it should include the property when serializing/deserializing. However, the myProperty property’s getter and setter does not follow the standard naming convention and will be ignored by Jackson unless annotated.

Although we do not need all the annotations, marking standard getter/setter methods with @JsonProperty makes the code more human-readable, and so we leave the annotations. Of course, again, do not update the generated classes; if you wish to modify the generated classes, then change the associated schema and regenerate the classes.

**CatalogImage POJO**

The plugin also generated a CatalogImage consisting of an identifier, path to the binary file, and metadata. The identifier and path are both strings, while the metadata is a list of CatalogMetaDatum objects.

**package *com.bts.imageclient.rest.api.types*;**

**import *java.util.ArrayList*;**

**import *java.util.List*;**

**import *com.fasterxml.jackson.annotation.JsonInclude*;**

**import *com.fasterxml.jackson.annotation.JsonProperty*;**

**import *com.fasterxml.jackson.annotation.JsonPropertyDescription*;**

**import *com.fasterxml.jackson.annotation.JsonPropertyOrder*;**

**/\*\***

**\* CatalogImage**

**\* <p>**

**\* A image from Image catalog**

**\***

**\*/**

**@JsonInclude(JsonInclude.Include.NON\_NULL)**

**@JsonPropertyOrder({**

**"image-id",**

**"image-format",**

**"image-path",**

**"meta-data"**

**})**

**public class CatalogImage {**

**/\*\***

**\* The unique identifier for an image.**

**\***

**\*/**

**@JsonProperty("image-id")**

**@JsonPropertyDescription("The unique identifier for an image.")**

**private String imageId;**

**/\*\***

**\* Format of image (gif, png, jpeg, etc.).**

**\***

**\*/**

**@JsonProperty("image-format")**

**@JsonPropertyDescription("Format of image (gif, png, jpeg, etc.).")**

**private String imageFormat;**

**/\*\***

**\* Path/URL to the image data.**

**\***

**\*/**

**@JsonProperty("image-path")**

**@JsonPropertyDescription("Path/URL to the image data.")**

**private String imagePath;**

**/\*\***

**\* Metadata item describing resource.**

**\***

**\*/**

**@JsonProperty("meta-data")**

**@JsonPropertyDescription("Metadata item describing resource.")**

**private List<CatalogMetaDatum> metaData = new ArrayList<CatalogMetaDatum>();**

**/\*\***

**\* The unique identifier for an image.**

**\***

**\*/**

**@JsonProperty("image-id")**

**public String getImageId() {**

**return imageId;**

**}**

**/\*\***

**\* The unique identifier for an image.**

**\***

**\*/**

**@JsonProperty("image-id")**

**public void setImageId(String imageId) {**

**this.imageId = imageId;**

**}**

**/\*\***

**\* Format of image (gif, png, jpeg, etc.).**

**\***

**\*/**

**@JsonProperty("image-format")**

**public String getImageFormat() {**

**return imageFormat;**

**}**

**/\*\***

**\* Format of image (gif, png, jpeg, etc.).**

**\***

**\*/**

**@JsonProperty("image-format")**

**public void setImageFormat(String imageFormat) {**

**this.imageFormat = imageFormat;**

**}**

**/\*\***

**\* Path/URL to the image data.**

**\***

**\*/**

**@JsonProperty("image-path")**

**public String getImagePath() {**

**return imagePath;**

**}**

**/\*\***

**\* Path/URL to the image data.**

**\***

**\*/**

**@JsonProperty("image-path")**

**public void setImagePath(String imagePath) {**

**this.imagePath = imagePath;**

**}**

**/\*\***

**\* Metadata item describing resource.**

**\***

**\*/**

**@JsonProperty("meta-data")**

**public List<CatalogMetaDatum> getMetaData() {**

**return metaData;**

**}**

**/\*\***

**\* Metadata item describing resource.**

**\***

**\*/**

**@JsonProperty("meta-data")**

**public void setMetaData(List<CatalogMetaDatum> metaData) {**

**this.metaData = metaData;**

**}**

**@Override**

**public String toString() {**

**StringBuilder sb = new StringBuilder();**

**sb.append(CatalogImage.class.getName()).append('@').append(Integer.toHexString(System.identityHashCode(this))).append('[');**

**sb.append("imageId");**

**sb.append('=');**

**sb.append(((this.imageId == null)?"<null>":this.imageId));**

**sb.append(',');**

**sb.append("imageFormat");**

**sb.append('=');**

**sb.append(((this.imageFormat == null)?"<null>":this.imageFormat));**

**sb.append(',');**

**sb.append("imagePath");**

**sb.append('=');**

**sb.append(((this.imagePath == null)?"<null>":this.imagePath));**

**sb.append(',');**

**sb.append("metaData");**

**sb.append('=');**

**sb.append(((this.metaData == null)?"<null>":this.metaData));**

**sb.append(',');**

**if (sb.charAt((sb.length()- 1)) == ',') {**

**sb.setCharAt((sb.length()- 1), ']');**

**} else {**

**sb.append(']');**

**}**

**return sb.toString();**

**}**

**@Override**

**public int hashCode() {**

**int result = 1;**

**result = ((result\* 31)+((this.imageFormat == null)? 0 :this.imageFormat.hashCode()));**

**result = ((result\* 31)+((this.metaData == null)? 0 :this.metaData.hashCode()));**

**result = ((result\* 31)+((this.imageId == null)? 0 :this.imageId.hashCode()));**

**result = ((result\* 31)+((this.imagePath == null)? 0 :this.imagePath.hashCode()));**

**return result;**

**}**

**@Override**

**public boolean equals(Object other) {**

**if (other == this) {**

**return true;**

**}**

**if ((other instanceof CatalogImage) == false) {**

**return false;**

**}**

**CatalogImage rhs = ((CatalogImage) other);**

**return (((((this.imageFormat == rhs.imageFormat)||((this.imageFormat!= null)&&this.imageFormat.equals(rhs.imageFormat)))&&((this.metaData == rhs.metaData)||((this.metaData!= null)&&this.metaData.equals(rhs.metaData))))&&((this.imageId == rhs.imageId)||((this.imageId!= null)&&this.imageId.equals(rhs.imageId))))&&((this.imagePath == rhs.imagePath)||((this.imagePath!= null)&&this.imagePath.equals(rhs.imagePath))));**

**}**

**}**

**CatalogClient POJO**

Recall that we left the CatalogClient.schema.json schema as an exercise and only created a bare-bones schema. The CatalogClient POJO generated is likewise minimal and included so we can later code a CatalogClientController without compile errors.

**package *com.bts.imageclient.rest.api.types*;**

**import *com.fasterxml.jackson.annotation.JsonInclude*;**

**import *com.fasterxml.jackson.annotation.JsonPropertyOrder*;**

**/\*\***

**\* CatalogClient**

**\* <p>**

**\* A Client from Image catalog**

**\***

**\*/**

**@JsonInclude(JsonInclude.Include.NON\_NULL)**

**@JsonPropertyOrder({**

**})**

**public class CatalogClient {**

**@Override**

**public String toString() {**

**StringBuilder sb = new StringBuilder();**

**sb.append(CatalogClient.class.getName()).append('@').append(Integer.toHexString(System.identityHashCode(this))).append('[');**

**if (sb.charAt((sb.length()- 1)) == ',') {**

**sb.setCharAt((sb.length()- 1), ']');**

**} else {**

**sb.append(']');**

**}**

**return sb.toString();**

**}**

**@Override**

**public int hashCode() {**

**int result = 1;**

**return result;**

**}**

**@Override**

**public boolean equals(Object other) {**

**if (other == this) {**

**return true;**

**}**

**if ((other instanceof CatalogClient) == false) {**

**return false;**

**}**

**CatalogClient rhs = ((CatalogClient) other);**

**return true;**

**}**

**}**

**REST Controllers**

Now that we have the resources representative as POJOs, let’s code the controllers, beginning with the CatalogImageController class.

**Catalog Image Controller**

Refer to the numbered list of REST endpoints above. We now implement those REST endpoints using Java and the JAX-RS 2.0 specification. The endpoints numbered 2, 4, 9, 11, and 13 are all related to manipulating an image, and so we implement them in CatalogImageController.

*Note that we implement the methods to return CatalogImage and CatalogClient objects, later we explore how these methods are returning JAX-RS Response objects (after serialized to JSON).*

* Create a new package com.bts.imageclient.rest.api.controller.
* Add a new class, CatalogImageController, to the newly created package.
* Add the endpoint methods without implementing the methods. For the method bodies, simply return a null.

**package *com.bts.imageclient.rest.api.controller*;**

**import *java.util.ArrayList*;**

**import *java.util.List*;**

**import *javax.ws.rs.DELETE*;**

**import *javax.ws.rs.GET*;**

**import *javax.ws.rs.PATCH*;**

**import *javax.ws.rs.PUT*;**

**import *javax.ws.rs.Path*;**

**import *javax.ws.rs.PathParam*;**

**import *javax.ws.rs.Produces*;**

**import *javax.ws.rs.QueryParam*;**

**import *javax.ws.rs.core.Response*;**

**import *org.springframework.stereotype.Component*;**

**import *com.bts.imageclient.rest.api.types.CatalogImage*;**

**import *com.bts.imageclient.rest.api.types.CatalogMetaDatum*;**

**@Component**

**@Path("/images")**

**@Produces("application/json")**

**@Api(value = "CatalogImageController Resource", produces = "application/json")**

**public class CatalogImageController {**

**@GET**

**@Path("/{image-id}")**

**public Response getImage(@PathParam("image-id") String imageId){return null;}**

**@GET**

**public List<CatalogImage> getImages(@QueryParam("meta-data") List<String> imageMetadata) {return null;}**

**@PUT**

**@Path("/{image-id}")**

**public CatalogImage replaceImage(@PathParam("image-id") String imageId, CatalogImage image) {return null;}**

**@PATCH**

**@Path("/{image-id}")**

**public CatalogImage updateImage(@PathParam("image-id") String imageId, CatalogImage image) {return null;}**

**@DELETE**

**@Path("/{image-id}")**

**public Response deleteImage(@PathParam("image-id") String imageId) throws CatalogImageDoesNotExistException {return null;}**

**}**

Each method is commented with the relevant corresponding interaction sentence determined during analysis and design. The annotations are what make this class special. The @Component annotation tells Spring that the class is a component and its life-cycle managed by Spring. The JAX-RS @Path annotation decorating the CatalogImageController class definition denotes the controller’s base path (/images). The @Path annotation denotes a method’s path when decorating a method definition. The @Path annotations determine the complete path to a resource. The @Produces annotation indicates the class returns JSON data. Note that you can override this behavior by decorating a method with its @Produces annotation. Each method has a corresponding annotation indicating the HTTP command used and, if applicable, an @PathParam annotation or QueryParam annotation identifying the path or query parameter. These annotations are what make this class a JAX-RS resource provider.

Note that we did not implement REST statements 5 or 7, despite their fetching images. Instead, we implement them as part of the client controller, which we code now.

**Catalog Client Controller**

The client controller implements endpoints 1, 3, 5, 6, 7, 8, 10, and 12.

* Create a new class named CatalogClientController in the com.bts.imageclient.rest.api.controller package.
* Add the endpoints applicable to clients, but do not implement the methods. As with the CatalogImageController, have each method return null.

**package *com.bts.imageclient.rest.api.controller*;**

**import *java.util.List*;**

**import *javax.ws.rs.DELETE*;**

**import *javax.ws.rs.GET*;**

**import *javax.ws.rs.PATCH*;**

**import *javax.ws.rs.POST*;**

**import *javax.ws.rs.PUT*;**

**import *javax.ws.rs.Path*;**

**import *javax.ws.rs.PathParam*;**

**import *javax.ws.rs.Produces*;**

**import *javax.ws.rs.QueryParam*;**

**import *javax.ws.rs.core.Response*;**

**import *org.springframework.stereotype.Component*;**

**import *com.bts.imageclient.rest.api.types.CatalogClient*;**

**import *com.bts.imageclient.rest.api.types.CatalogImage*;**

**@Component**

**@Path("/clients")**

**@Produces("application/json")**

**public class CatalogClientController {**

**@GET**

**@Path("/{client-id}")**

**public CatalogClient getClient(@PathParam("client-id") String clientId){return null;}**

**@GET**

**public List<CatalogClient> getClients(@QueryParam("client-metadata") List<String> clientMetadata){return null;}**

**@GET**

**@Path("/{client-id}/images")**

**public List<CatalogImage> getImages(@PathParam("client-id") String clientId, @QueryParam("image-metadata") List<String> imageMetadata){return null;}**

**@POST**

**public CatalogClient createClient(CatalogClient client){return null;}**

**@PUT**

**@Path("/{client-id}")**

**public CatalogClient replaceClient(@PathParam("client-id") String clientId, CatalogClient client){return null;}**

**@PATCH**

**@Path("/{client-id}")**

**public CatalogClient updateClient(@PathParam("client-id") String clientId, CatalogClient client){return null;}**

**@DELETE**

**@Path("/{client-id}")**

**public Response delete(@PathParam("client-id") String clientId){return null;}**

**@POST**

**@Path("/clients/{client-id}/images")**

**public CatalogImage createImage(@PathParam("client-id") String clientId){return null;}**

**}**

**Create JerseyConfig File**

To access our newly created JAX-RS resources, we need to register them as Jersey resources. An easy way to accomplish this is by creating a Jersey configuration file.

* Create a new package named com.bts.imageclient.config and add a class called JerseyConfig to the package.

**package *com.bts.imageclient.config*;**

**import *javax.annotation.PostConstruct*;**

**import *javax.ws.rs.ApplicationPath*;**

**import *org.glassfish.jersey.server.ResourceConfig*;**

**import *org.glassfish.jersey.server.wadl.internal.WadlResource*;**

**import *org.springframework.beans.factory.annotation.Value*;**

**import *org.springframework.stereotype.Component*;**

**import *com.bts.imageclient.rest.api.controller.CatalogClientController*;**

**import *com.bts.imageclient.rest.api.controller.CatalogImageController*;**

**@Component**

**@ApplicationPath("/imageclient")**

**public class JerseyConfig extends ResourceConfig {**

**@PostConstruct**

**public void init() {**

**configEndPoints();**

**}**

**private void configEndPoints(){**

**register(CatalogClientController.class);**

**register(CatalogImageController.class);**

**}**

**}**

The @Component annotation tells Spring to manage this class. The @ApplicationPath annotation tells our JAX-RS implementation, Jersey, that the base URL for the application is imageclient. At the same time, the @PostConstruct tag instructs Spring to execute the init method after instantiating an instance of JerseyConfig. The ResourceConfig class is Jersey specific and implements the JAX-RS Application interface. The application, and by definition the ResourceConfig, defines the application’s JAX-RS components.

* Compile the application using Maven and correct any errors.

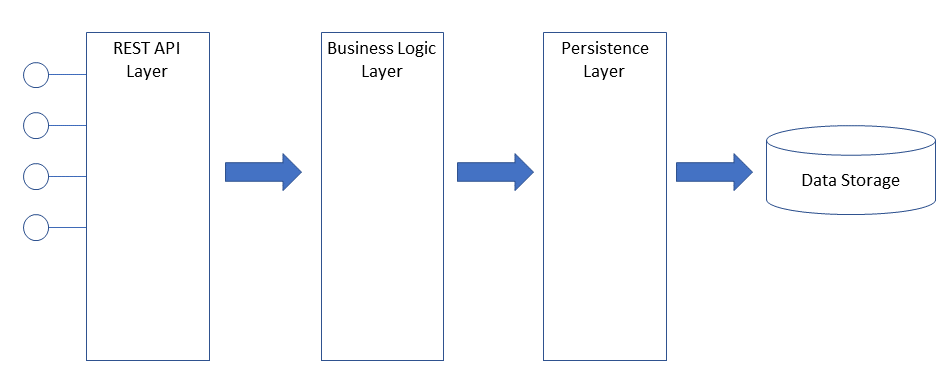
We are not testing anything yet, as we did not implement any controller methods. Let’s implement several of the methods in CatalogImageController, starting with the getImages and getImage REST endpoints. The methods only contain throw-away code that illustrates we called the methods correctly.

**Implement getImages and getImage Methods**

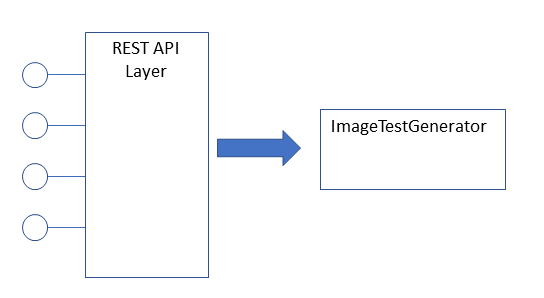
Before implementing the getImages and getImage methods, we need to have data to return from the REST endpoints. However, as we have not implemented the underlying application, we create a simple class that generates test data instead of the real thing.

**Code a Test Data Generator**

In a real-world project, you would connect the REST API layer to another layer containing your project’s business logic, which would connect to a persistence layer. For instance, the following diagram illustrates a typical web architecture.



The Multitier architecture page on Wikipedia ([Multitier architecture](https://en.wikipedia.org/wiki/Multitier_architecture)) is a good starting point for more information on a tiered architecture. But here, we do not concern ourselves with the entire application, but rather, only the REST API layer, so instead of implementing a full application, we implement a much simpler architecture, using a class called ImageTestGenerator to generate test data.

API using a test data generator.

* Create a package named com.bts.imageclient.rest.api.types.util and create a new class named ImageTestGenerator.
* Implement the generateTestImages and generateTestImage methods as static methods. Note the methods simply generate test data for the tutorial and have nothing to do with a “REST API.”

**package *com.bts.imageclient.rest.api.types.util*;**

**import *java.util.ArrayList*;**

**import *java.util.List*;**

**import *com.bts.imageclient.rest.api.types.CatalogImage*;**

**import *com.bts.imageclient.rest.api.types.CatalogMetaDatum*;**

**public class CatalogImageTestGenerator {**

**public static List<CatalogImage> generateTestImages()**

**{**

**List<CatalogImage> images = new ArrayList<CatalogImage>();**

**for(int i = 0; i < 20; i++) {**

**CatalogImage image = new CatalogImage();**

**List<CatalogMetaDatum> metaData = new ArrayList<CatalogMetaDatum>();**

**for(int k = 0; k < 10; k++) {**

**CatalogMetaDatum item = new CatalogMetaDatum();**

**item.setName("name"+k);**

**item.setValue("value" + k);**

**metaData.add(item);**

**}**

**image.setImageId(Integer.toString(i));**

**image.setMetaData(metaData);**

**image.setImageFormat("png");**

**image.setImagePath("http://www.nowhere.com/image-binary-url/image" + i + "." + image.getImageFormat());**

**images.add(image);**

**}**

**return images;**

**}**

**public static CatalogImage generateTestImage(StringimageId) {**

**CatalogImage image = new CatalogImage();**

**List<CatalogMetaDatum> metaData = new ArrayList<CatalogMetaDatum>();**

**for(int k = 0; k < 10; k++) {**

**CatalogMetaDatum item = new CatalogMetaDatum();**

**item.setName("name"+k);**

**item.setValue("value" + k);**

**metaData.add(item);**

**}**

**image.setImageId(imageId.toString());**

**image.setMetaData(metaData);**

**image.setImageFormat("png");**

**image.setImagePath("http://www.nowhere.com/image-binary-url/image" + imageId + ".png");**

**return image;**

**}**

**}**

**Implement the getImages and getImage Methods**

Let’s implement the getImages and getImage methods so we can illustrate calling the REST endpoints. As previously mentioned, we implement the methods with throw-away code that merely allows us to verify we called the methods correctly.

**Write the Code**

* Add code to getImages to obtain imageMetadata as a list and print each list item. Have the system then return a list of CatalogImage objects by calling CatalogImageTestGenerator’s generateTestImages method.
* Modify getImage to return an image by calling the CatalogImageTestGenerator’s generateTestImage method.

**@GET**

**public List<CatalogImage> getImages(@QueryParam("meta-data") List<String> imageMetadata)**

**{**

**List<CatalogMetaDatum> metaData = imageMetadata.stream().map(c -> {**

**String[] nameValue = c.split("=");**

**CatalogMetaDatum item = new CatalogMetaDatum();**

**item.setName(nameValue[0]);**

**item.setValue(nameValue[1]);**

**return item;**

**}).collect(Collectors.toList());**

**metaData.stream().forEach(c->System.out.println(c.toString()));**

**List<CatalogImage> returnImages = CatalogImageTestGenerator.generateTestImages();**

**return returnImages;**

**}**

**@GET**

**@Path("/{image-id}")**

**public CatalogImage getImage(@PathParam("image-id") String imageId)**

**{**

**return CatalogImageTestGenerator.generateTestImage(imageId);**

**}**

The getImages method now takes a list of metadata query parameters and prints those parameters. Remember that when passing multiple values for the same query parameter, you reuse the name, as the following URL illustrates.

**http://localhost:8080/imageclient/images?meta-data=name1=value1&meta-data=name2=value2&meta-data=name3=value3**

The method then returns a list of CatalogImage objects. Of course, Jackson converts the list and its objects to the appropriate JSON.

The getImage method takes a single CatalogImage’s unique identifier as a path parameter and returns the image. Of course, in a real application, it would fetch that image from the application’s data layer; however, here we simply create a test CatalogImage.  The URL, if the resource’s id were “123” would appear as follows.

**http://localhost:8080/imageclient/images/123**

* Compile and package the application using Maven.
* Start the application so we can test using Postman.

**Test Using Postman**

Postman is a powerful tool used to test web services freely downloaded from its website ([postman.com](https://www.postman.com/)) on the download page ([download Postman](https://www.postman.com/downloads/)). If you have never used Postman before, you should refer to the tutorials at the postman learning website ([Postman Learning Center](https://learning.postman.com/)). However, you should be able to follow along, even if you are not familiar with Postman.

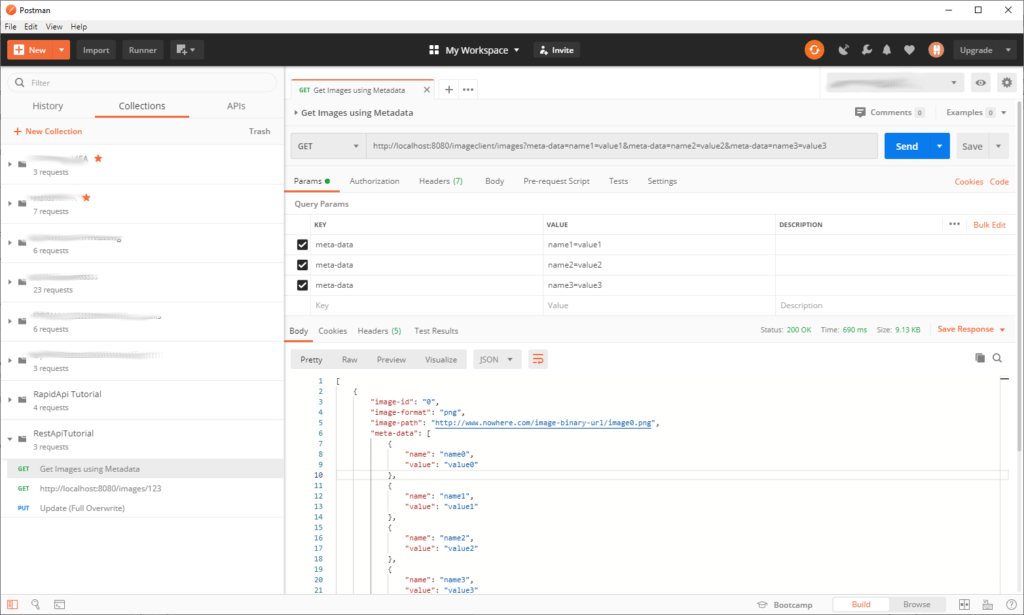
* Create a new GET request that calls the images URL.
* Create the following three Query Params.

|  |  |
| --- | --- |
| Query Params Added to GET Request | |
| meta-data | name1=value1 |
| meta-data | name2=value2 |
| meta-data | name3=value3 |

After adding the query params, the following URL is visible in the URL field in Postman.

**http://localhost:8080/imageclient/images?meta-data=name1=value1&meta-data=name2=value2&meta-data=name3=value3**

* Click Send, and Postman requests the images from our REST API.

Postman requesting CatalogImage objects.

If everything works correctly, the response appears in the Body results tab. The response consists of a JSON array of ten CatalogImage resources, and each resource contains ten MetaDatum resources.

**[**

**{**

**"image-id": "0",**

**"image-format": "png",**

**"image-path": "http://www.nowhere.com/image-binary-url/image0.png",**

**"meta-data": [**

**{**

**"name": "name0",**

**"value": "value0"**

**},**

**< ---- snip ---- >**

**{**

**"name": "name9",**

**"value": "value9"**

**}**

**]**

**},**

**<--- snip ---->**

**{**

**"image-id": "19",**

**"image-format": "png",**

**"image-path": "http://www.nowhere.com/image-binary-url/image19.png",**

**"meta-data": [**

**{**

**"name": "name0",**

**"value": "value0"**

**},**

**<--- snip ---->**

**{**

**"name": "name9",**

**"value": "value9"**

**}**

**]**

**}**

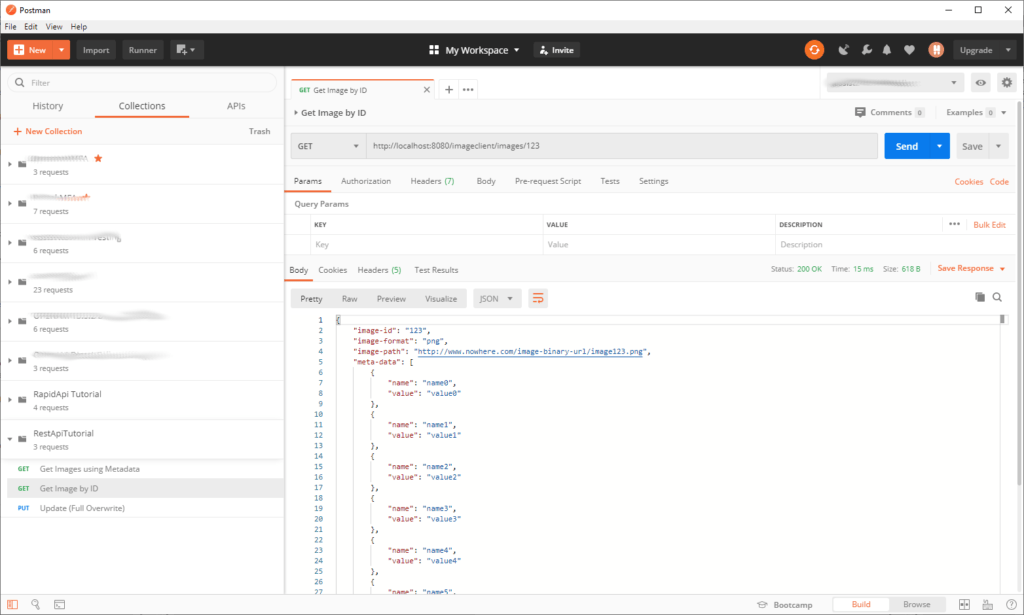
**]**

Now test getting a CatalogImage by its identifier.

* Create a new GET request adding “123” as the identifier.  The URL should appear as follows.

**http://localhost:8080/imageclient/images/123**

* Click Send, and the response consists of a single CatalogImage.



Postman requesting a particular CatalogImage.

**Implement replaceImage and updateImage**

Now that we have implemented two GET requests of resources, let’s implement the methods that use PUT and PATCH to replace and update a CatalogImage.  Note that we do not implement creating a CatalogImage in the CatalogImageController class. Instead, we placed that method in the CatalogClient class.  This decision is debatable; however, because an image cannot exist independent of a client, it seems logical to implement that functionality in the CatalogClientController.  But replacing an image and updating an image is not reliant upon a CatalogClient; both actions are performed directly on an image with a particular identifier, and so are included in the CatalogImageController.

**Write the Code**

* Implement the replaceImage method.
* To ensure the method is working, simply change the CatalogImage’s path.

**@PUT**

**@Path("/{image-id}")**

**public CatalogImage replaceImage(@PathParam("image-id") String imageId, CatalogImage image) {**

**System.out.println(image.toString());**

**image.setImagePath("http://www.nowhere.com/changed-path.jpg");**

**return image;**

**}**

The throw-away code simply takes a CatalogImage, prints it, and then changes the image’s path so we can ensure the path was changed by reviewing the response. Remember, PUT overwrites a resource, not update, but suspend disbelief and pretend our code replaces a CatalogImage and then returns that CatalogImage.

* Implement the updateImage method.

**@PATCH**

**@Path("/{image-id}")**

**public CatalogImage updateImage(@PathParam("image-id") String imageId, CatalogImage image){**

**System.out.println(image.toString());**

**image.setImageId("http://www.nowhere.com/patch-path.jpg");**

**CatalogMetaDatum val = image.getMetaData().get(0);**

**val.setName("patch");**

**val.setValue("patch-value");**

**List<CatalogMetaDatum> theList = new ArrayList<CatalogMetaDatum>();**

**theList.add(val);**

**image.setMetaData(theList);**

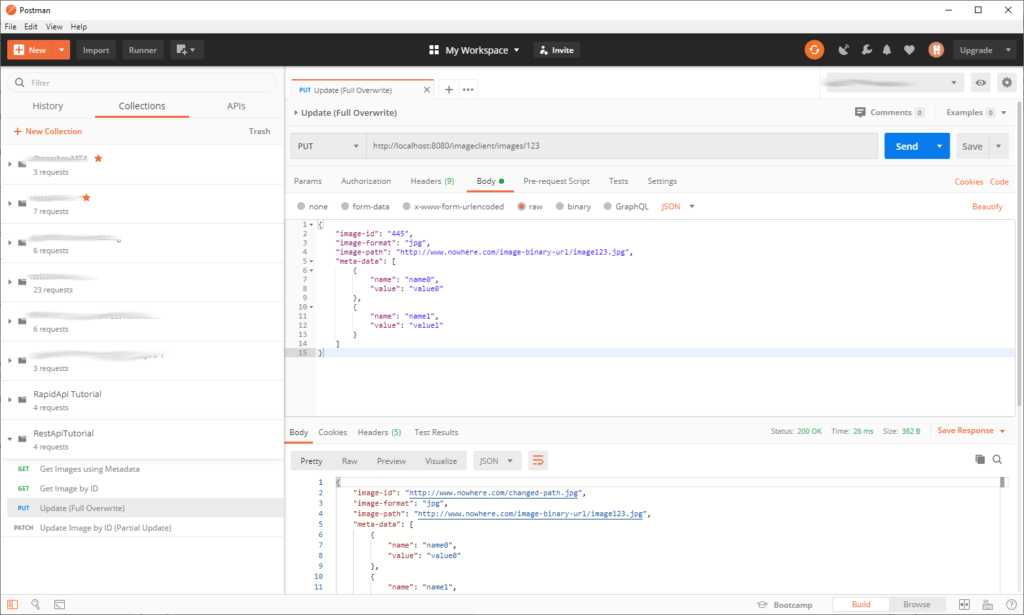
**return image;**

**}**

* Compile and package the application using Maven.
* Start the application so you can test it using Postman.

**Test Using Postman**

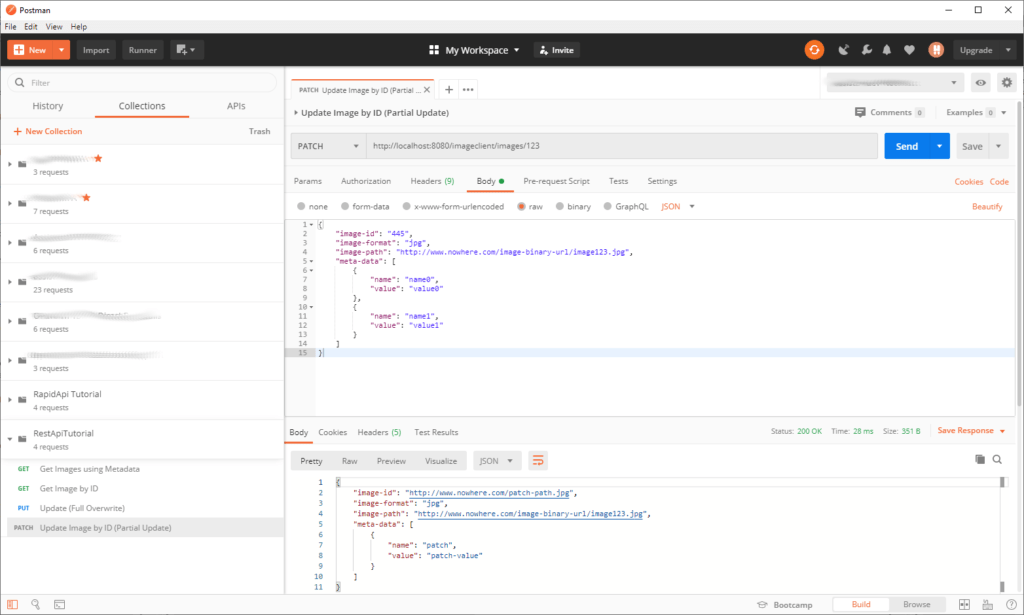
* Create a new PUT request in Postman with “123” as the URL parameter.
* Click Send, and the endpoint returns the CatalogImage object.



Postman entirely replacing a CatalogImage using PUT.

Of course, the method’s implementation was hypothetical. In reality, it would replace the CatalogImage in an application’s data store with the CatalogImage passed as a parameter.

* Create a new PATCH request in Postman with “123” as the URL parameter.
* Click send, and the endpoint returns the CatalogImage object.



Postman partially updating a CatalogImage using PATCH.

The PATCH method was also hypothetical.  Recall that PATCH updates an existing resource with the modified properties; it does not replace the existing resource.

**Modify getImage method**

We now modify the getImage method to return a JAX-RS Response object. It is essential to realize that the RESTful API methods are not returning the objects, but JSON representations of a Response object deserialized to a JSON string.

**JAX-RS Response**

A Response is an abstract class provided by the JAX-RS specification. This class is what a JAX-RS endpoint returns. But note, throughout this tutorial, we returned Java objects; for example, getImages returned a list of CatalogImage objects. But this is an illusion; what the methods returned is a serialized Response object. Behind the scenes, the Jackson library automatically converts the custom object to a Response, which Jersey then serializes into an HTTP response. This automatic conversion allows developers to take a shortcut and return POJOs from methods implementing REST endpoints. Let’s illustrate this automatic conversion by modifying getImage to return a Response.

**Write the Code**

* Change the getImage method to return a Response. Note that it uses the builder pattern to construct the response.
* Build and package the application using Maven.

**@GET**

**@Path("/{image-id}")**

**public Response getImage(@PathParam("image-id") String imageId)**

**{**

**CatalogImage theImage = CatalogImageTestGenerator.generateTestImage(imageId);**

**Response resp = Response.status(Response.Status.OK).entity(theImage).build();**

**return resp;**

**}**

* Start the application to allow testing with Postman.

The method loads the CatalogImage into the response and then returns the response rather than the CatalogImage.

**Test Using Postman**

* Perform the same steps as taken earlier to test the getImage method.
* Note that the response in Postman is the same as when the method returned a CatalogImage.

**Implement delete Method**

Now that we have discussed the Response object let’s implement the delete method.

**// 12. Delete the client with the specified client-id.**

**@DELETE**

**@Path("/{client-id}")**

**public Response delete(@PathParam("client-id") String clientId) {return null;}**

Note that it returns a Response object. Think about it, if you delete a CatalogImage, then you shouldn’t return that deleted CatalogImage. Instead, return a simple success string.

**{"CatalogImage deleted successfully."}**

**Write the Code**

* Implement the delete method to return a simple success message.

**@DELETE**

**@Path("/{image-id}")**

**public Response deleteImage(@PathParam("image-id") String imageId) {**

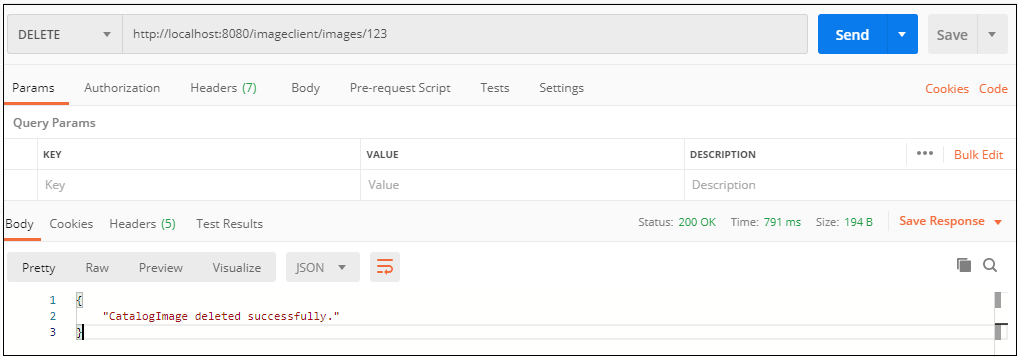
**return Response.ok().entity("{"CatalogImage deleted successfully."}").build();**

**}**

* Build and package the application using Maven.
* Start the application so we can test it using Postman.

**Test Using Postman**

* Create a new DELETE method in Postman.
* Click Send, and the JSON message appears in the body after calling the delete endpoint.



Deleting an image in Postman.

**Documenting the API with Swagger**

Learning about a REST API’s endpoints through trial and error using a tool like Postman is slow and error-prone; It’s much better to provide explicit documentation to a client. But rather than delivering manually written documentation, it would be much better if you could provide documentation that also allowed developers to interact with your API. Enter Swagger. Swagger is a tool that can generate robust documentation, available through a URL on the same server as your API. Moreover, developers can test a REST APIs endpoints using Swagger.

Swagger implements the OpenAPI specification (OAS). You can access the OpenAPI specification at its website ([OpenAPI Specification](https://swagger.io/specification/v2/)).  Let’s set up our project to use Swagger. Remember, our Spring Boot project uses Jersey JAX-RS and not Spring’s MVC; therefore, setting up Swagger is different than you will read on most Spring Boot tutorials on the web.

Modify POM and JerseyConfig

Supporting Swagger requires modifying our project’s POM and JerseyConfig files. Also, add the spring-boot-starter-web to the POM so we can display the generated Swagger documentation as web pages.

* Add the swagger-jersey2-jaxrs dependency to the POM.
* Add the spring-boot-starter-web dependency to the POM.

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-web</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>io.swagger</groupId>**

**<artifactId>swagger-jersey2-jaxrs</artifactId>**

**<version>1.5.9</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-web</artifactId>**

**</dependency>**

* Modify JerseyConfig to configure Swagger.

**package *com.bts.imageclient.config*;**

**import *javax.annotation.PostConstruct*;**

**import *javax.ws.rs.ApplicationPath*;**

**import *org.glassfish.jersey.server.ResourceConfig*;**

**import *org.glassfish.jersey.server.wadl.internal.WadlResource*;**

**import *org.springframework.beans.factory.annotation.Value*;**

**import *org.springframework.stereotype.Component*;**

**import *com.bts.imageclient.rest.api.controller.CatalogClientController*;**

**import *com.bts.imageclient.rest.api.controller.CatalogImageController*;**

**import *io.swagger.jaxrs.config.BeanConfig*;**

**import *io.swagger.jaxrs.listing.ApiListingResource*;**

**import *io.swagger.jaxrs.listing.SwaggerSerializers*;**

**@Component**

**@ApplicationPath("/imageclient")**

**public class JerseyConfig extends ResourceConfig {**

**@PostConstruct**

**public void init() {**

**configureSwagger();**

**configEndPoints();**

**}**

**private void configEndPoints(){**

**register(CatalogClientController.class);**

**register(CatalogImageController.class);**

**}**

**private void configureSwagger() {**

**register(ApiListingResource.class);**

**register(SwaggerSerializers.class);**

**register(WadlResource.class);**

**BeanConfig config = new BeanConfig();**

**config.setTitle("ImageClient Tutorial API");**

**config.setHost("localhost:8080");**

**config.setVersion("v1");**

**config.setContact("Joe Programmer");**

**config.setSchemes(new String[] { "http", "https" });**

**config.setBasePath("/imageclient");**

**config.setResourcePackage("com.bts.imageclient.rest.api.controller");**

**config.setPrettyPrint(true);**

**config.setScan(true);**

**}**

**}**

* Add the Swagger @Api annotation to the CatalogImageController class.

**@Component**

**@Path("/images")**

**@Produces("application/json")**

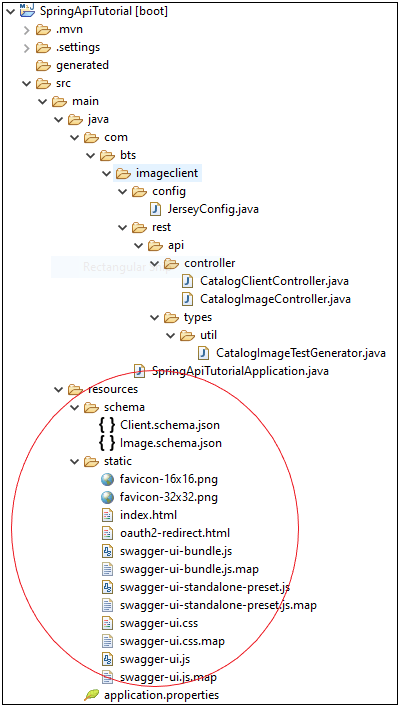
**@Api(value = "CatalogImageController Resource", produces = "application/json")**

**public class CatalogImageController {**

Swagger UI

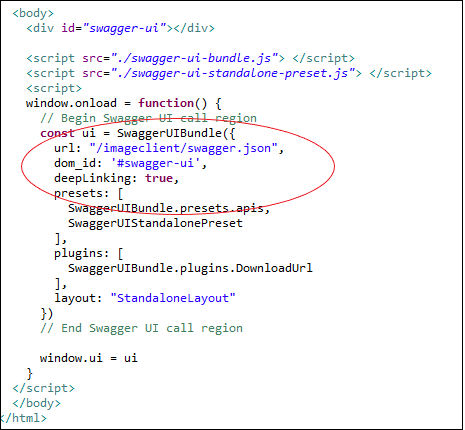
Swagger generates documentation as a JSON file. However, to display the API as dynamic web pages, we must download and add Swagger UI to our project. You can download Swagger UI from its website ([swagger-ui](https://swagger.io/tools/swagger-ui/)).

* Download Swagger-UI and unzip the file.
* Create a folder named static in your project’s resources folder.
* Copy the content of the swagger-ui dist folder to your project’s static folder.



Swagger-UI content in the Spring Boot resources folder.

* Open the index.html file and replace the URL with the path to the swagger.json file (we set that path in the JerseyConfig file).



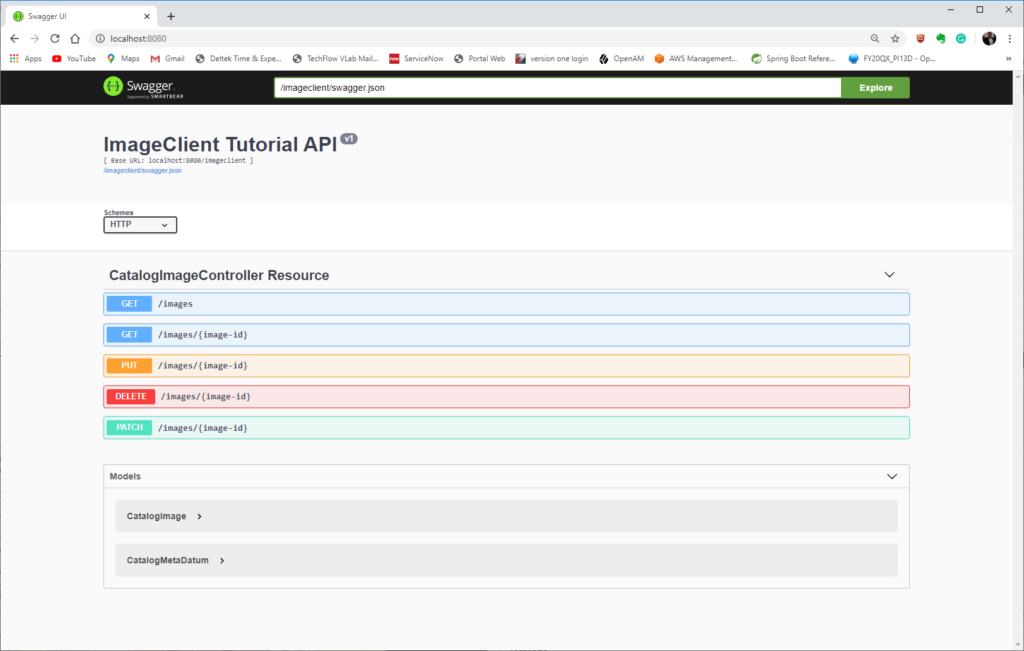
Swagger-UI index.html file.

View API In Swagger-UI

* Build and package the project using Maven.
* Start the application.
* Navigate to your application’s root directory in a web browser.

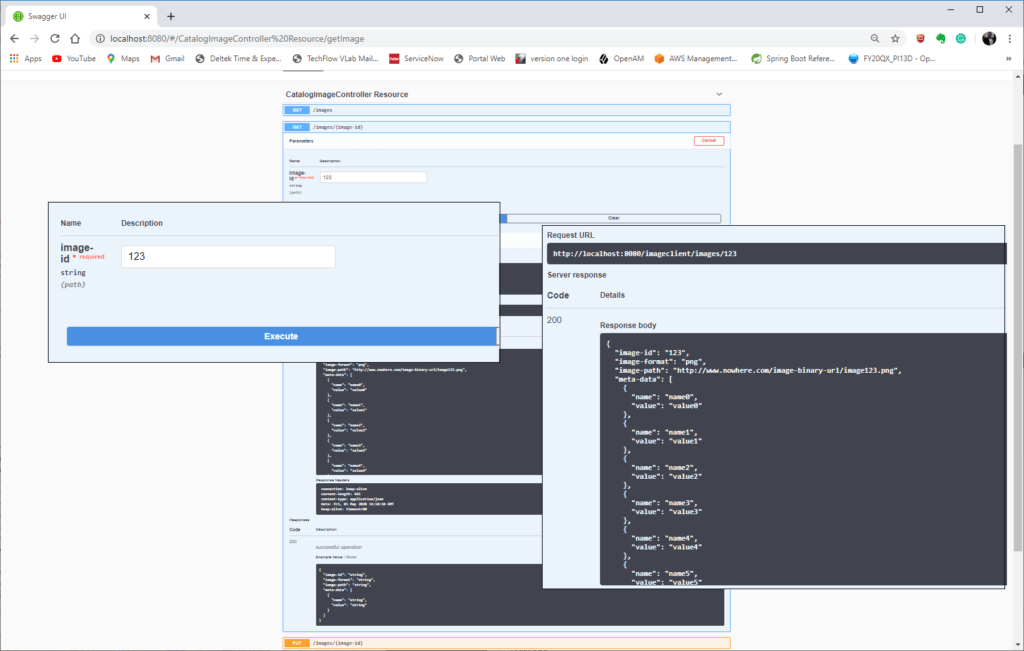
**http://localhost:8080/**

If you configure Swagger correctly, you should see the CatalogImageController Resource documentation.



The CatalogImage in Swagger.

* Press the GET /images/{image-id} link, and expand the section.
* Click the Try It Out button and enter a number in image-id.
* Press the Execute button, and Swagger-UI returns the response.



Swagger exercising an endpoint directly.

Swagger not only documents the API, but it also provides an interface to explore your API with actual values. Swagger’s documentation allows developers of client applications to review the API documentation and interactively learn how to use the API. Note that Swagger-UI also shows the curl command if you wish to use curl to access the API.

**curl -X GET "http://localhost:8080/imageclient/images/123" -H "accept: application/json"**

Swagger is a powerful tool to document your API. Note that we only added the @Api annotation to the CatalogImageController class, but Swagger has many more available annotations that you might use to generate documentation of your API. Refer to the Swagger website for a complete listing of available annotations ([Open API 2.X Annotations](https://github.com/swagger-api/swagger-core/wiki/Swagger-2.X---Annotations)). A “getting started” guide is available from the Swagger Github repository ([Swagger 2.X Getting Started](https://github.com/swagger-api/swagger-core/wiki/Swagger-2.X---Getting-started)).

**Status Codes & Exceptions**

Oops, this tutorial made the same mistake many developers make when coding; we only considered the happy path where everything works.  What happens when things go awry?  Let’s end this tutorial by briefly discussing REST API exceptions. These exceptions differ from regular Java exceptions, as a REST endpoint consumes an HTTP request and returns an HTTP response. Part of the returned response is a response code.  A RESTful API should always return the proper response code.

* *Note that this tutorial uses HTTP status code and HTTP response code as synonyms. Although status is more commonly used than response, both are used to refer to the codes returned with an HTTP response.*

HTTP Status Codes

HTTP Status codes indicate if a request was handled correctly.  There are five groups of response codes.

|  |  |  |
| --- | --- | --- |
| HTTP Response Code Categories | | |
| 100-199 | Informational |  |
| 200-299 | Success | Client’s request successfully processed. |
| 300-399 | Redirection | The client must take additional action to complete the request. |
| 400-499 | Client Error | An error caused by the client occurred. |
| 500-599 | Server Error | An error caused by the server occurred. |

The most commonly used HTTP Status codes encountered are as follows.

|  |  |  |
| --- | --- | --- |
| 200 | OK | Successful GET, PUT, PATCH, DELETE. |
| 201 | Created | Successful POST. |
| 204 | No Content | No response body sent. |
| 400 | Bad Request | Request not understood due to malformed syntax. |
| 401 | Unauthorized | No authentication provided or authentication fails. |
| 403 | Forbidden | Authentication succeeded, but the client has no access to resources. |
| 404 | Not Found | Non-existent resource requested. |
| 415 | Unsupported Media Type | The request contained an incorrect content type. |
| 500 | Internal Server Error | General error on the server. |
| 503 | Service Unavailable | Server unavailable. |

A complete listing of HTTP status codes is available on the Wikipedia page ([List of HTTP status codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)).

A RESTful API should return the appropriate error codes. And do not return a 200 response code with an error as the response’s body no matter how great the temptation due to expediency (more common than you think). If you take a little time to return meaningful errors and error codes, your API will prove much easier to use. Now that we have examined HTTP status codes let’s see how these codes are used with JAX-RS exceptions.

JAX-RS Exceptions

The JAX-RS exception handling strategy provides an easy way to map exceptions in your code to the appropriate HTTP responses.  The top-level exception class JAX-RS provides is named the WebApplicationException, and its subclasses are ClientErrorException, RedirectionException, and ServerErrorException. The WebApplicationException constructor, unlike a more standard Java exception, can take an integer as an HTTP status code. Refer to the java documentation for more detail ([JavaDoc](https://docs.oracle.com/javaee/7/api/javax/ws/rs/WebApplicationException.html" \t "_blank)).

Although JAX-RS provides custom exceptions, a more straightforward approach is to use an ExceptionMapper to map the error to a response. This mapping allows you to develop custom responses for an exception. In the following example, we use an ExceptionMapper, but before we do, we first implement a simple custom exception to illustrate the ExceptionMapper’s functionality fully.

**Create an Example Business Exception**

Earlier, this tutorial admonished you to treat the business layer as a black-box.  However, to illustrate exception handling accurately, we must stray from that assumption. Usually, business logic results in business errors.  For example, if our back-end system attempted to delete a non-existent CatalogImage, it might throw a CatalogImageDoesNotExist exception. Of course, this is just a hypothetical example, but the critical point is that systems usually throw custom exceptions.

* Create a new package named com.bts.exceptions and create a new class named CatalogImageDoesNotExistException that extends Exception.
* Compile and package the application using Maven.

**package *com.bts.exception*;**

**public class CatalogImageDoesNotExistException extends Exception {**

**}**

**JAX-RS ExceptionMapper**

The ExceptionMapper interface maps exceptions to Response objects. When the application throws an exception of the mapped type, the mapper automatically handles the exception and returns a Response.

* Create the class CatalogImageDoesNotExistExceptionMapper and place it in the com.bts.imageclient.config package.
* Implement the ExceptionMapper interface.
* Implement the toResponse interface method.

**package *com.bts.imageclient.config*;**

**import *javax.ws.rs.core.MediaType*;**

**import *javax.ws.rs.core.Response*;**

**import *javax.ws.rs.ext.ExceptionMapper*;**

**import *javax.ws.rs.ext.Provider*;**

**import *com.bts.exception.CatalogImageDoesNotExistException*;**

**@Provider**

**public class CatalogImageDoesNotExistExceptionMapper implements ExceptionMapper<CatalogImageDoesNotExistException> {**

**@Override**

**public Response toResponse(CatalogImageDoesNotExistException exception) {**

**return Response.status(Response.Status.NOT\_FOUND).type(MediaType.APPLICATION\_JSON).entity("{"The CatalogImage does not exist."}").build();**

**}**

**}**

The method implementation, admittedly simplistic, returns a simple JSON response with the 404 HTTP response code and a simple JSON message informing the caller that the method CatalogImage was not found.

**Modify RestConfig**

* Modify RestConfig.java to register the CatalogImageDoesNotExistExceptionMapper.class. Although the @Provider JAX-RS annotation allows JAX-RS to discover this class when scanning annotations and register it as an exception mapping provider, keep it simple and register the class manually in the RestConfig class.

**@PostConstruct**

**public void init() {**

**register(CatalogImageDoesNotExistExceptionMapper.class);**

**configureSwagger();**

**configEndPoints();**

**}**

Modifying the API

Now that we have mapped the exception let’s modify the delete method to throw this exception. However, before we do, pause to consider what happens if your code does not handle an exception. As you will see from the example, JAX-RS automatically translates the error into a generic error that returns a 500 HTTP response code and a JSON error message.

**Modify getImage**

* Modify getImage so it generates a null pointer exception when passing “foo” as the image identifier.

**@GET**

**@Path("/{image-id}")**

**public CatalogImage getImage(@PathParam("image-id") String imageId)**

**{**

**if(imageId.equals("foo"))**

**{**

**CatalogMetaDatum x = null;**

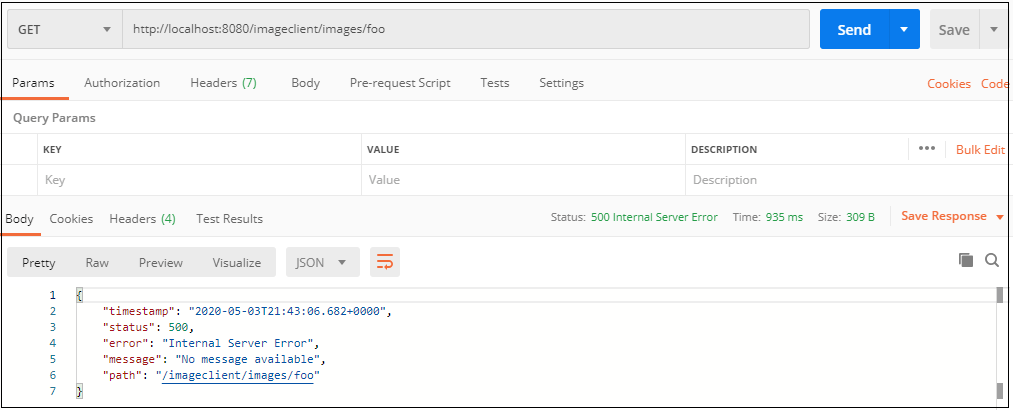
**x.setName("foo");**

**}**

**return CatalogImageTestGenerator.generateTestImage(imageId);**

**}**

* Test in Postman and note the method returns the error as the response and 500 as the HTTP status code.



The default exception returned by Jersey.

JAX-RS returned a status code of 500 and a generic error JSON response for us.  So if you forget error handling, you can rest assured that clients will receive an HTTP status code indicating an error occurred, and an error message provided as JSON.

**Modify delete Method**

Assume that when the application attempts to delete a non-existent CatalogImage, it throws a CatalogImageNotFoundException. To mimic this behavior, modify deleteImage to throw this exception.

* Modify deleteImage so it throws a CatalogImageDoesNotExistException if the CatalogImage’s identifier is “foo.”

**@DELETE**

**@Path("/{image-id}")**

**public Response deleteImage(@PathParam("image-id") String imageId) throws CatalogImageDoesNotExistException {**

**if(imageId.equals("foo")) throw new CatalogImageDoesNotExistException();**

**return Response.ok().entity("{"CatalogImage deleted successfully."}").build();**

**}**

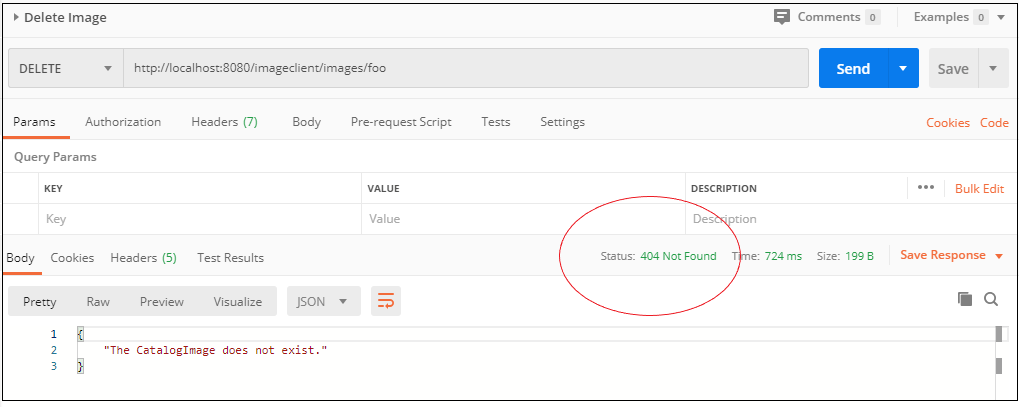
* Compile and package the application using Maven.
* Start the application so we can test it using Postman.

**Test in Postman**

* Modify the call to delete so that the identifier is “foo.”

**http://localhost:8080/imageclient/images/foo**

* Click Save and the Response returned is the string mapped in the Mapper and is 404 response code.



Deleting a resource that does not exist.

Mapping business exceptions to responses allows RESTful APIs to send meaningful errors to client software. Map exceptions to responses so that your RESTful API returns useful HTTP status codes and error messages.

**API Versioning**

Versioning is often an afterthought when developing a RESTful API. This oversight can lead to problems later as, after having created a RESTful API, how can an API be versioned without significant refactoring? Fortunately, there is at least a straightforward technique to add versioning to a preexisting API.  Let’s use this technique to add versioning to our API. The technique is to add a version to our URL and to extend ClientImageController with a child class.  Note that this technique is not a good technique for APIs that are changed frequently; however, a well designed RESTful API should not require change often.

Modify CatalogImageController & JerseyConfig

An easy way to modify your API is by creating a subclass of your original controller. The subclass implements the modified methods while its parent (the original controller) remains unchanged.

* Create a new package com.bts.imageclient.rest.api.controller.v2
* Create a new class named CatalogImageControllerV2 that extends CatalogImageController.
* Change the @Path annotation to include v2 in the path.
* Modify getImage to return a simple string to confirm it is being called and not getImage in CatalogImageController.

**package *com.bts.imageclient.rest.api.controller.v2*;**

**import *javax.ws.rs.GET*;**

**import *javax.ws.rs.Path*;**

**import *javax.ws.rs.PathParam*;**

**import *javax.ws.rs.Produces*;**

**import *javax.ws.rs.core.Response*;**

**import *org.springframework.stereotype.Component*;**

**import *com.bts.imageclient.rest.api.controller.CatalogImageController*;**

**import *io.swagger.annotations.Api*;**

**@Component**

**@Path("/v2/images")**

**@Produces("application/json")**

**@Api(value = "CatalogImageControllerV2 Resource", produces = "application/json")**

**public class CatalogImageControllerV2 extends CatalogImageController {**

**@GET**

**@Path("/{image-id}")**

**public Response getImage(@PathParam("image-id") String imageId)**

**{**

**return Response.ok().entity("{"This is version Two."}").build();**

**}**

**}**

* Modify JerseyConfig to register the new controller.

**private void configEndPoints(){**

**register(CatalogClientController.class);**

**register(CatalogImageController.class);**

**register(CatalogImageControllerV2.class);**

**}**

* Compile and package the application using Maven.
* Start the application so we can test in Postman.

Test in Postman

* Open Postman and get an image using the original URL.

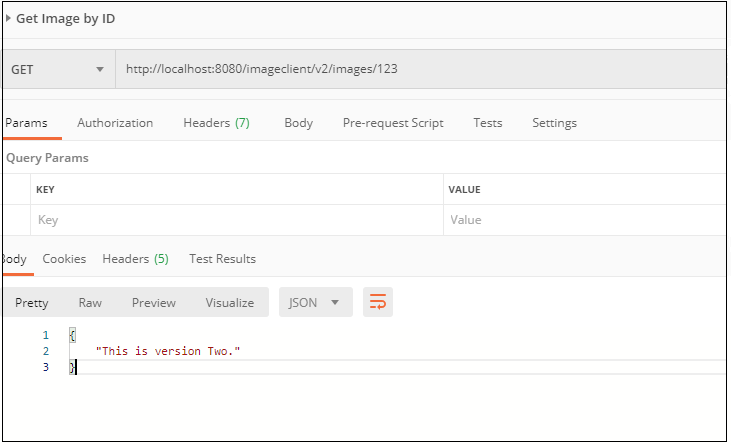
**http://localhost:8080/imageclient/images/123**

As expected, Postman invoked the original REST endpoint and returned a CatalogImage as JSON.

* Add v2 to the path to get an image using the new API version.

**http://localhost:8080/imageclient/v2/images/123**

Postman returns the JSON message from the new version of getImage rather than the parent class getImage version.

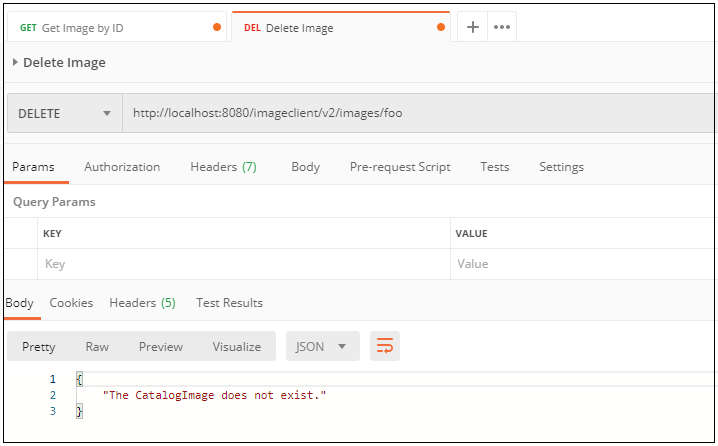


Calling version two of our API.

* Add v2 to the delete URL and click Send.

**http://localhost:8080/imageclient/v2/images/foo**

Postman still returns the original response expected from delete.

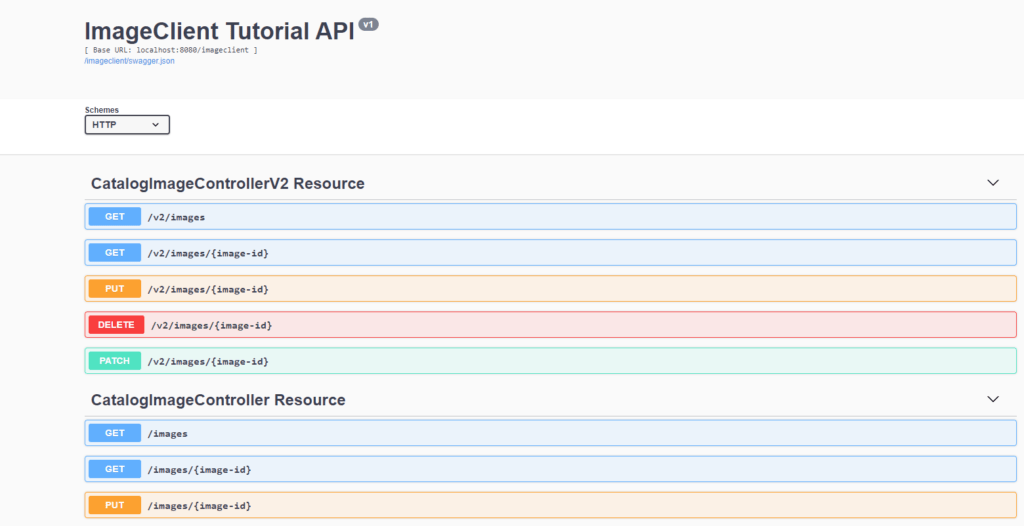


Calling v2 of delete still calls original delete endpoint.

Calling the original delete, even though we added v2 to the URL, makes sense, as the delete method, only exists in CatalogImageController. The original method, in the parent class CatalogImageController, was called because deleteImage was not overridden by CatalogImageControllerV2.

Review in Swagger

* Open the API’s Swagger UI.



Swagger displayed both API versions.

As this simple example illustrates, adding a version to an API’s base URL is an easy means of adding versioning to an API.

**Publishing on RapidAPI**

Whew, we covered a lot of material. But we have one more thing to cover; publishing your API on RapidAPI.  After all, you are reading this on RapidAPI’s blog, so naturally, this tutorial should end by publishing the API on RapidAPI. But seriously, if your API is for inter-departmental communication, then you probably do not want to publish it as a public API. But, let’s suppose your API is public. Just as nobody will read a tutorial, if he or she cannot find it, nobody will use your API if they cannot find it.  RapidAPI is a centralized location where people go to find published API. By publishing your API to RapidAPI, your API has a much higher chance of being discovered by potential users. So let’s end by publishing the API to RapidAPI.

Deploying API to a Server

We will not discuss how to publish your API to a web server. However, we do briefly explain Heroku ([Heroku website](https://www.heroku.com/)), which is what I used to host my Spring Boot application for this tutorial. The following two resources should be sufficient to deploy the API to Heroku if you wish to use Heroku. In particular, if you follow the tutorial listed in the second bullet, you will duplicate the steps I took to publish the API.

* The documentation on deploying Spring Boot to Heroku is available here ([Deploying Spring Boot Applications to Heroku](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku)).
* This resource is a good tutorial explaining the steps to publish a Spring Boot REST API to Heroku ([Create and Publish Your Rest API Using Spring Boot and Heroku](https://dzone.com/articles/spring-boot-heroku-and-cicd)).
* *Before deploying your application to a server, do not forget to change the Swagger host in your JerseyConfig.java file.*

**private void configureSwagger() {**

**register(ApiListingResource.class);**

**register(SwaggerSerializers.class);**

**register(WadlResource.class);**

**BeanConfig config = new BeanConfig();**

**config.setTitle("ImageClient Tutorial API");**

**config.setHost("https://imagetestcatalog.herokuapp.com");**

**config.setVersion("v1");**

**config.setContact("Joe Programmer");**

**config.setSchemes(new String[] { "http", "https" });**

**config.setBasePath("/imageclient");**

**config.setResourcePackage("com.bts.imageclient.rest.api.controller");**

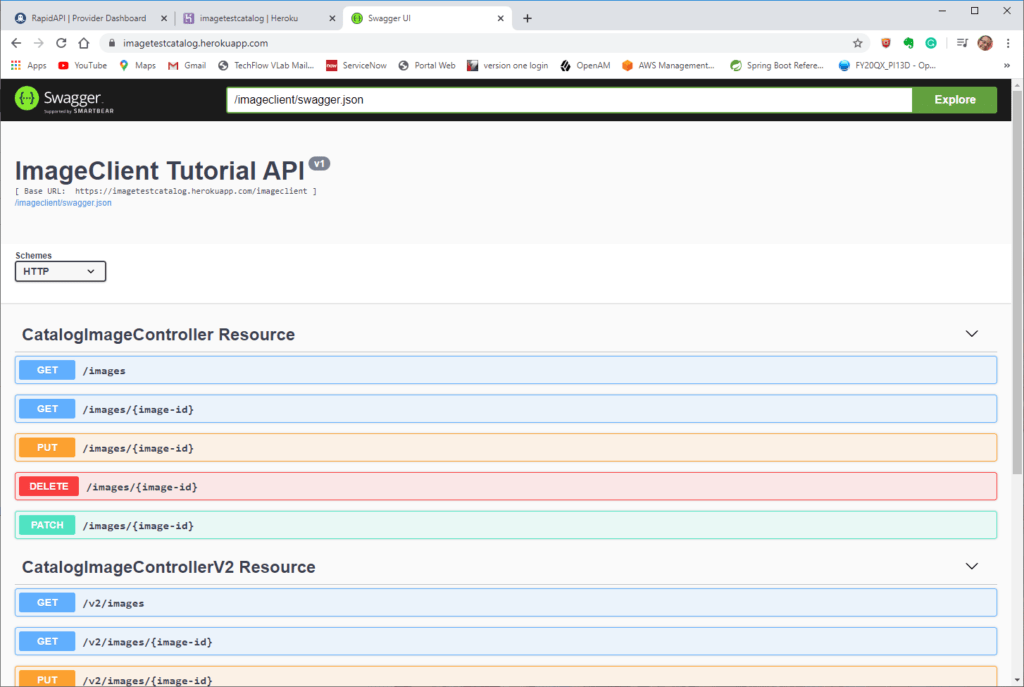
**config.setPrettyPrint(true);**

**config.setScan(true);**

**}**

After deploying your application, you should be able to navigate to the Swagger API. Let’s continue assuming you deployed the application to Heruku.

* Navigate to the Swagger UI for your site, for me it was https://imagetestcatalog.herokuapp.com.
* Notice the Base URL and the path to the swagger.json file directly below the “ImageClient Tutorial API” heading.
* Right-click on the URL and download the file.
* If you wish, test a few of the endpoints the same way we did earlier in the tutorial.

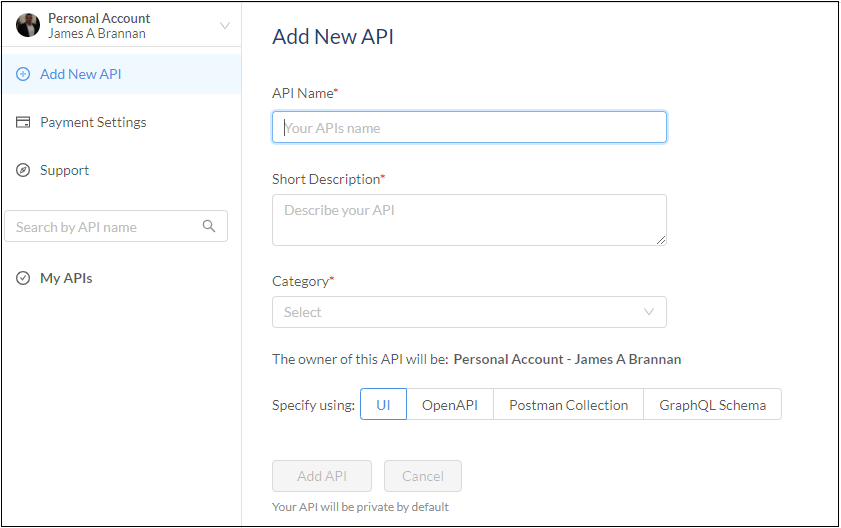


Application’s Swagger in Heroku.

Publishing API on RapidAPI

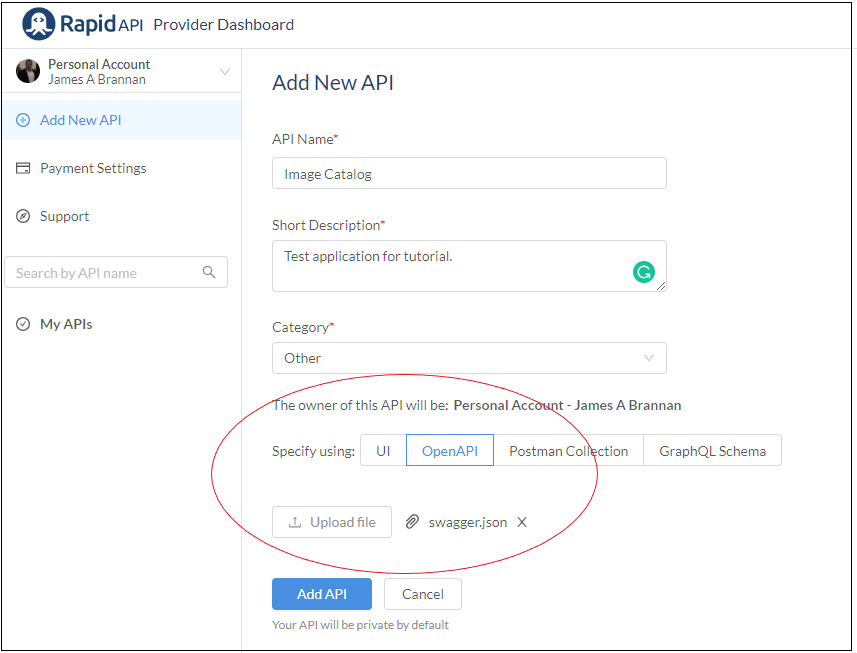
After publishing the API to Heroku, the next step is publishing the API to RapidAPI. Rapid API has ample documentation on getting started publishing your API ([Add an API](https://docs.rapidapi.com/docs/getting-started)), and it is assumed here that you already have a RapidAPI account and have familiarity using RapidAPI. Before continuing, read the documentation outlining adding an API to RapidAPI. After reading the documentation, complete the following steps to add the API to RapidAPI (changing names to your own).

* Click on the Add New API to open the Add New API screen.



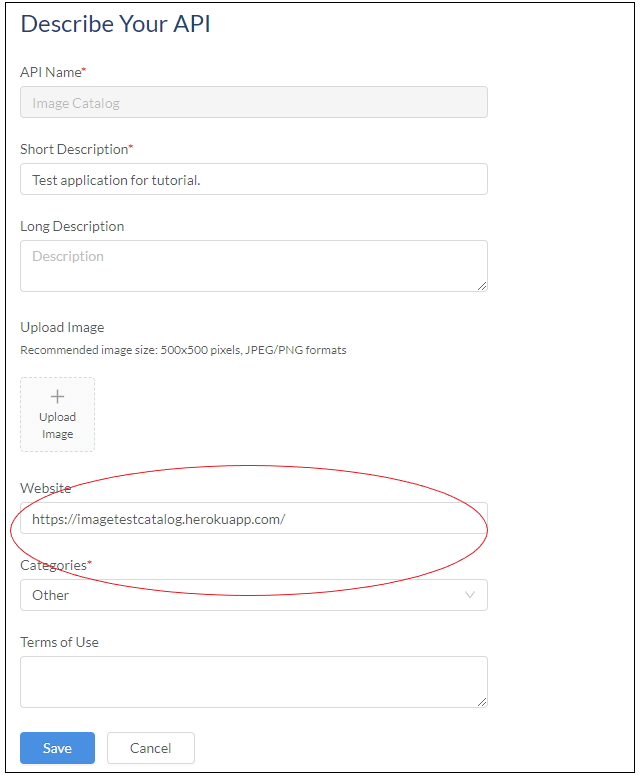
Add a new API on RapidAPI.

* Add an API Name and Short Description.
* Click on OpenAPI and upload the swagger.json file you saved locally. Be sure to use the Swagger file from where you published your API and not when we developed locally.
* Click Add API after uploading the Swagger file.



Uploading Swagger file to define API.

* On the next screen, add the website name and click save.

Add API base URL to RapidAPI.

* Navigate to your profile page and notice RapidAPI created yourAPI.

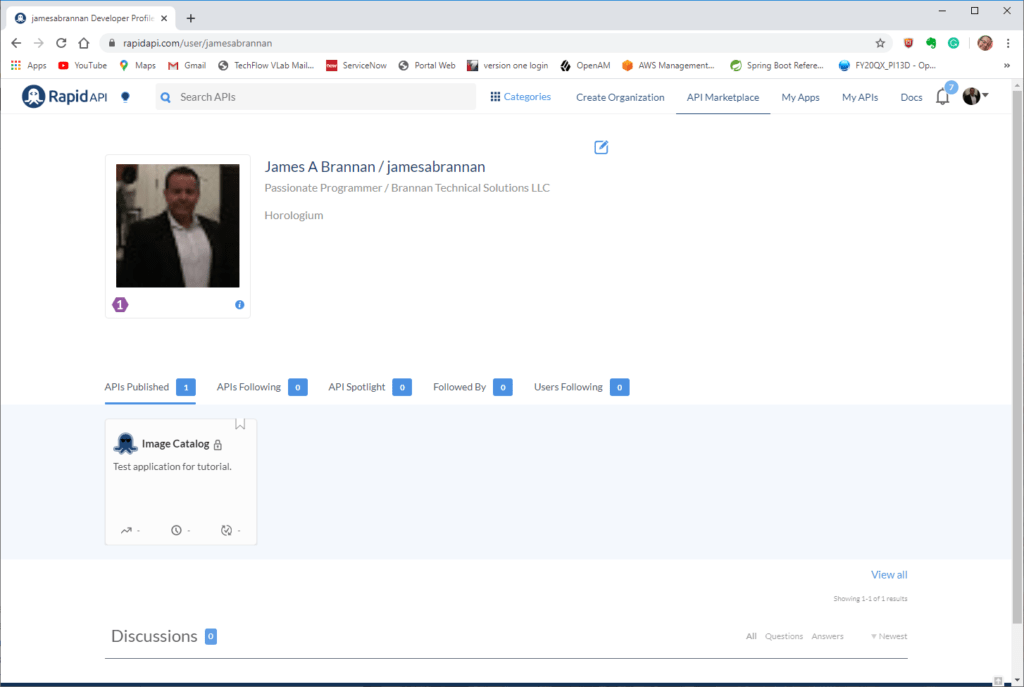


Image Catalog API visible on the profile page.

* Click on the API, and RapidAPI takes you to your API’s documentation page.

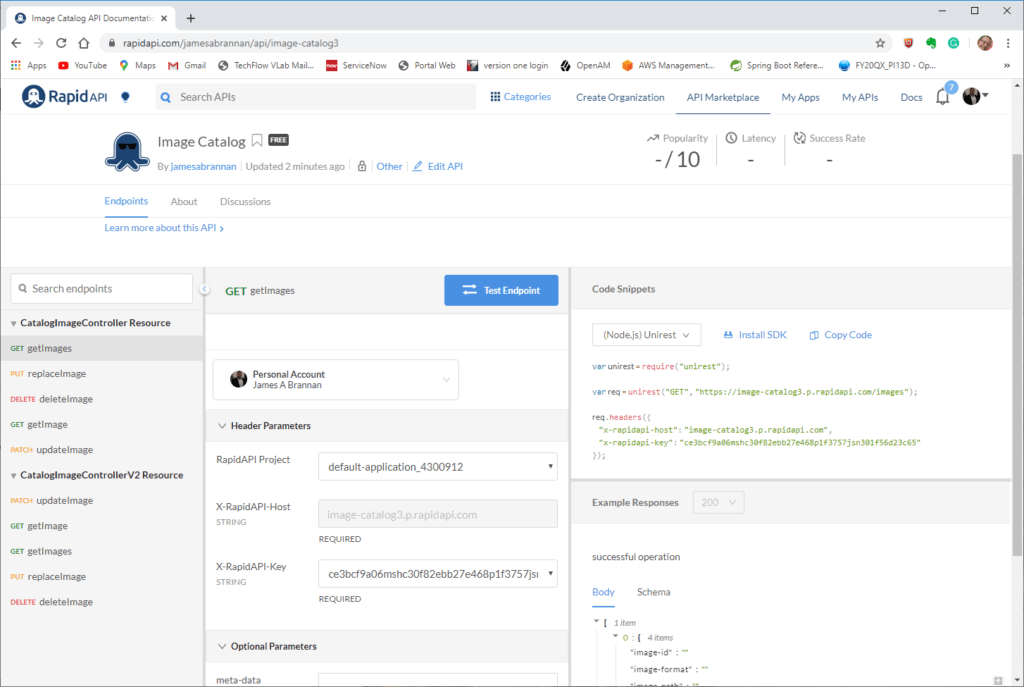
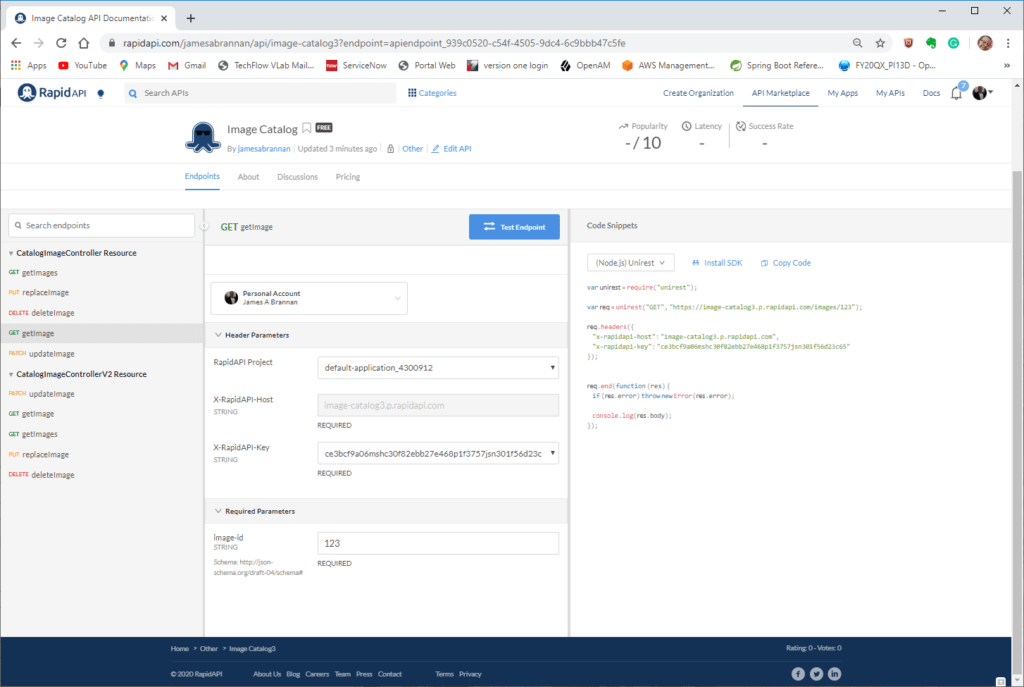


Image Catalog API on RapidAPI.

* *Note that the API is private, you must make your API public for others to find and use your API (not discussed here).*

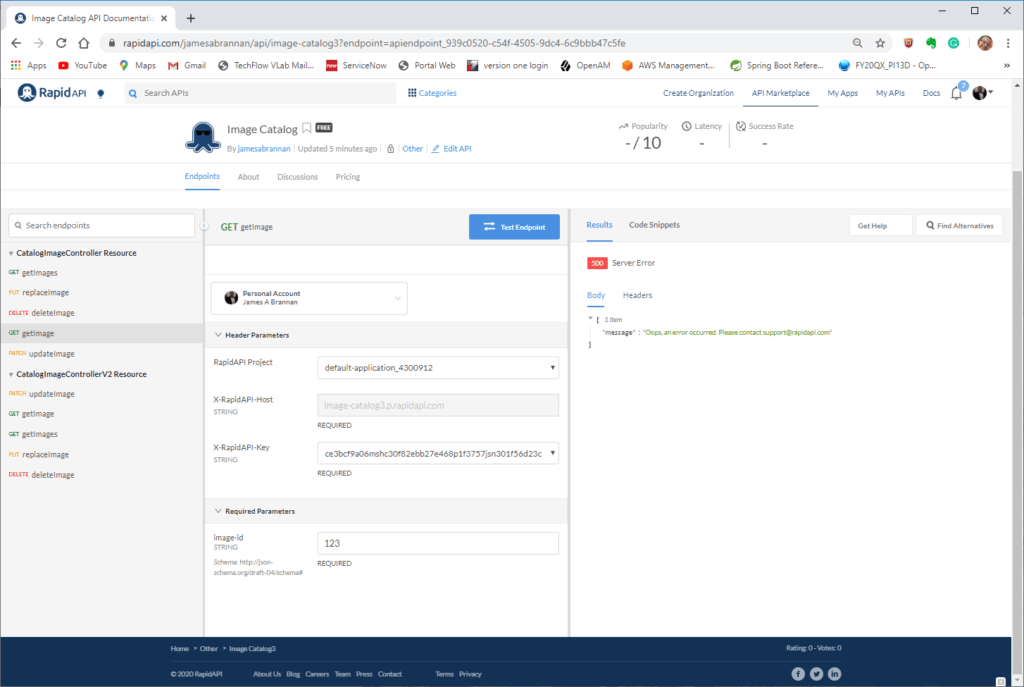
Let’s test that the API is configured correctly by calling one of its endpoints.

* Test the getImage endpoint by entering the value “123” into the image-id and then clicking Test Endpoint.



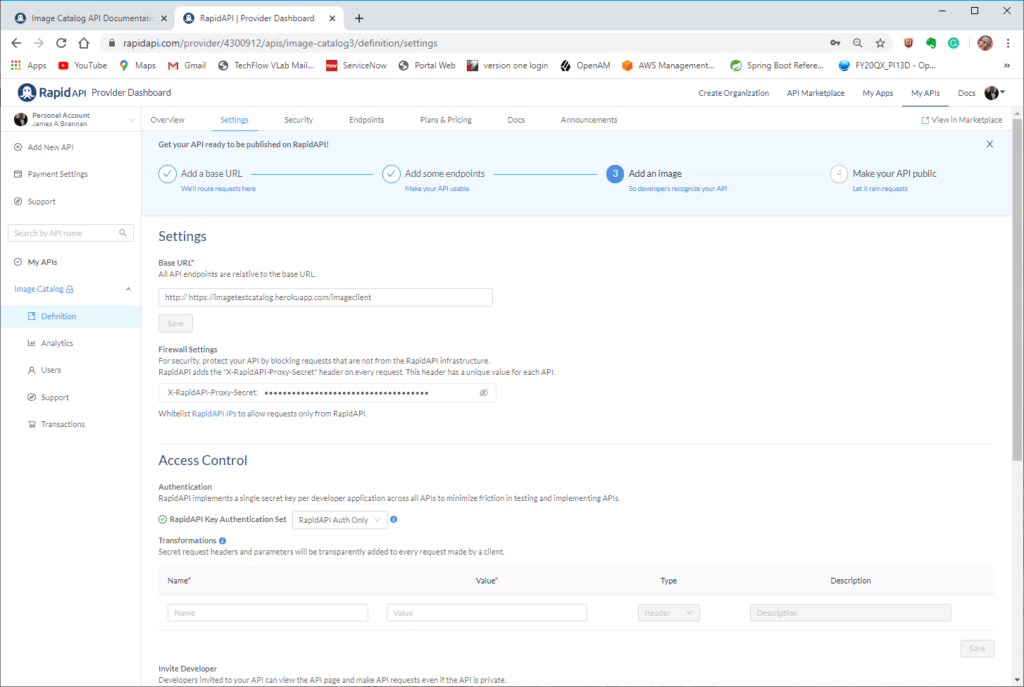
Testing getImage on RapidAPI.

* If, upon testing, you discover an error, do not panic.  Note that I received a Server Error when calling my endpoint.



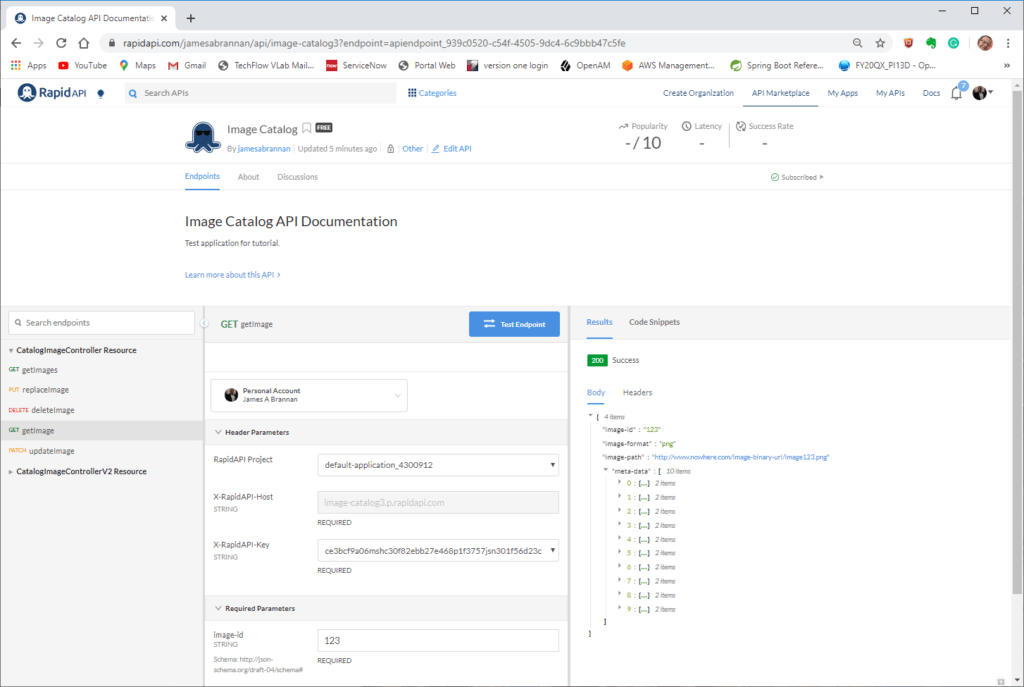
Calling getImage endpoint resulted in error.

* Navigate to your API under My APIs and click Edit.
* Click on the Settings tab.
* Fix any errors.



Fix the Base URL if in error.

Notice in the settings above, the Base URL is incorrect. So I fixed the URL and returned to my API documentation page. I then tested the getImage endpoint and obtained the expected response.



Testing the getImage endpoint and receiving the correct result.

Although the following steps are sufficient for development, refer to the documentation for complete instructions on publishing your API on RapidAPI. There you will find more information on the settings, security, endpoints, plans & pricing, and other information needed to publish your API. Remember, your API need not be free; you can charge users and get paid through RapidAPI for your API.  For more information, refer to the RapidAPI documentation.

**Summary**

We covered a lot of material in this tutorial. First, we designed a RESTful API. We then covered some best practices to follow while designing that API. Although design, in particular formal design, has lost the appeal it once had, do not skip the design tasks listed in this tutorial, even if your design is nothing more than pencil markings on a napkin. Remember, finding and fixing a bug on paper is a magnitude easier than finding and fixing that bug in code. Model the resources and, if necessary, formalize the resultant resource model into a UML object model. We finalized our API design by determining the list of interactions an external system will perform with our API. After designing our API, we implemented our API using Jersey running in a Spring Boot application. We then documented our API using Swagger. But then, after noticing we overlooked exception handling and versioning, we added both to our API. After implementing our RESTful API, we deployed it to a web server and then published it on RapidAPI.

Developing a RESTful API is a lot of work. But trust the extra effort spent carefully designing and implementing an API will pay dividends. You now have a better understanding of the process behind designing and implementing a RESTful API. Creating a robust and easy to use API ensures your application fulfills its client’s needs. And at the end of the day, that is the only important criterion for judging a successful API.

[**View the Best JavaScript APIs List**](https://rapidapi.com/collection/javascript-apis)

5/5 - (5 votes)

« [How To Create a React App Using Typescript](https://rapidapi.com/blog/create-react-app-typescript/)

[How to Integrate External API in a WordPress Page](https://rapidapi.com/blog/integrate-external-api-wordpress/) »

**Filed Under:**[**Java API Tutorials**](https://rapidapi.com/blog/category/tutorial/java/)**,**[**REST API Tutorials**](https://rapidapi.com/blog/category/tutorial/)**,**[**Spring Boot API Tutorials**](https://rapidapi.com/blog/category/tutorial/java/spring-boot/)**Tagged With:**[**create api**](https://rapidapi.com/blog/tag/create-api/)**,**[**how to build an api**](https://rapidapi.com/blog/tag/how-to-build-an-api/)**,**[**java**](https://rapidapi.com/blog/tag/java/)**,**[**spring boot**](https://rapidapi.com/blog/tag/spring-boot/)

A white circle on a gray background

Description automatically generated

**Kelly Arellano**

Top of Form

Search this website

Bottom of Form

Primary Sidebar

**Build anything with APIs, faster.**

Discover, evaluate, and integrate with any API. RapidAPI is the world’s largest API Hub with over 4 Million developers and 35,000 APIs.

[**Browse APIs »**](https://rapidapi.com/hub)

**APIs mentioned in this article**



[**Browse Most Popular APIs**](https://rapidapi.com/collection/popular-apis)

Footer

* [**API Guides**](https://rapidapi.com/guides)
* [**API Courses**](https://rapidapi.com/courses)
* [**API Glossary**](https://rapidapi.com/blog/api-glossary/)
* [**API Testing**](https://rapidapi.com/blog/api-testing/)
* [**API Management**](https://rapidapi.com/blog/api-management/)
* [**Most Popular APIs**](https://rapidapi.com/blog/most-popular-api/)
* [**Free APIs List**](https://rapidapi.com/collection/list-of-free-apis)
* [**How to use an API**](https://rapidapi.com/blog/how-to-use-an-api/)
* [**Learn REST API**](https://rapidapi.com/learn/rest)
* [**Build API’s**](https://rapidapi.com/blog/20-tutorials-on-how-to-create-your-own-api-sorted-by-programming-language/)
* [**About**](https://rapidapi.com/company/)
* [**Build APIs**](https://rapidapi.com/products/build-apis/)
* [**Careers**](https://rapidapi.com/company/careers/)
* [**Contact Us**](https://rapidapi.com/contact/)
* [**Write for Us**](https://rapidapi.com/blog/write-for-us/)
* [**API Directory**](https://rapidapi.com/blog/directory/)
* [**Press Room**](https://rapidapi.com/press/)
* [**Privacy Policy**](https://rapidapi.com/privacy/)
* [**Terms of Use**](https://rapidapi.com/terms/)

© 2024 RapidAPI. All rights reserved.